Inventing the Adventure Game

The Design of *Adventure* and *Rocky's Boots*

by Warren Robinett

An unpublished book manuscript from the dawn of the video-game age (written in 1983-84).

CONTENTS

Appendix A: Program Structure of *Adventure* and *Rocky's Boots*

Program Structure of Atari 2600 *Adventure*. Program Structure of *Rocky's Boots*.

About the games discussed in this book:

Collosal Cave Adventure was the first adventure game. It was completely text-based, and had no graphics at all. The player viewed text descriptions of the rooms and objects in the game world, and typed commands (such as "GO NORTH") to move through the game world. It was played on main-frame computers of the late 1970's.

Adventure, a video game cartridge for the *Atari 2600* video game console, was the first action-adventure video game. It was published by Atari Inc. in 1979, and sold 1 million copies.

Rocky's Boots was a commercial educational software product, written for the Apple II computer. It was published in 1982 by the Learning Company, and was one of the first educational simulations for home computers. It won Software-of-the-Year awards from *Learning* magazine (1983), *Parent's Choice* magazine (1983), and *Infoworld* magazine (1982, runner-up).

Preface

This book is about the thinking and design behind the invention of the first action-adventure game: *Adventure* for the Atari 2600 home video game console. It also covers the educational game *Rocky's Boots*, which was, in many ways, a sequel to *Adventure*.

I wrote the manuscript for this book in 1983-84. The idea was to write a design history of the two main game projects I had worked on over the previous five years: *Adventure* and *Rocky's Boots*. *Adventure* was the first action-adventure game and had sold 1 million copies. It also contained the first "Easter egg" in a computer game -- I had outfoxed Atari's management (who was trying to keep all their game designers anonymous) by hiding my signature in a hard-to-find secret room in the game world. The other game, *Rocky's Boots*, was one of the first educational simulations, and it won Software-of-the-Year awards from several magazines in 1982 and 1983. It was also the flagship product at the launch of the Learning Company, which was to become the dominant publisher of educational software within a few years.

Because both of these games got quite a bit of attention in their day, I thought that people -- especially those who wanted to design computer games themselves -- might be interested in the thought processes behind the final game designs.

At that time -- the early 1980's -- there were no old game designers because video games had only been around for a few years. As far as I knew, no one had ever written anything about the design of computer games. It seemed to me that although we would never be able to get a first-hand report about the invention of painting, or poetry, or song, or story-telling -- those innovators being lost in the mists of time -- that I could at least tell my part of the story about the creation of the new medium of computer games. And I thought I should do it right then, while the details were still fresh in my mind.

So I did. But by the time that I finished my manuscript in late 1984, Atari and the entire video game industry was collapsing. The publisher that was to have published my book, which specialized in computer titles, went out of business. So I was left with an unpublished manuscript, and by the time the world was again interested in video games -- thanks to Nintendo -- I doubted there would be much interest in a book about a ten-year-old video game. For many years, I thought no one was interested in the old Atari video games. But the Internet made it easier to find people a few years ago, and I began to hear from people interested in the old "classic" video games. A number of people requested that I publish my manuscript somehow, so I am now making it available on my website

http://www.WarrenRobinett.com

This site contains additional information (on emulators, maps, links to interviews, etc.) about *Adventure* and *Rocky's Boots*. To those of you who sent me Email: thanks for your interest, and here is the manuscript you asked for. I resisted the temptation to edit what I wrote in 1983-84, figuring the original vintage would be preferred. I may clean up the hand-drawn diagrams one of these days, but for now you get it unfiltered.

Looking back, I guess *Atari 2600 Adventure* was the ancestor of the *Legend of Zelda*, the *Ultima* series, and many other adventure and role-playing games. *Rocky's Boots* was the precursor to the "sim" genre, such as *SimCity*. If you want to hear how it all got started, read on.

Warren Robinett
Chapel Hill, North Carolina
March 2001

Chapter 1

The First Video Games

The first video games sprang up like wild mushrooms on rotten logs. There was no careful cultivation by the corporate, government, or research bureaucracies. The creators were young researchers, engineers, and graduate students, who began experimenting with early computer graphics because they were fascinated. There was something really interesting about controlling little moving patterns of light by pushing switches; it was new, it was unexplored, and it had possibilities. The instinct of man the explorer roared to life. And unlike the explorations of the academic researcher, where the excitement of discovering new continents of knowledge must be expressed in stacks of paper, there was an immediacy and engagingness to these computer-controlled magic light shows.

Where it was necessary, the bureaucracies were placated with the excuse that these games made good "demos." That was true. When a computer sat chewing invisibly on some complicated mathematical problem, it was about as interesting as an old refrigerator that no longer worked. "What does it do?" has always been a tough question for computer owners, and the pretty, flickering, responsive, mysterious display demonstrated that indeed there were fast and intricate things happening inside that dull gray box. The computer people already knew this, of course, but it was nice to be able to convince friends and dignitaries with 30 seconds of " "Look at this!" instead of a tedious, jargon-filled speech.

It is said that the best things in computer science have come about because someone said "I want it." Not the company president, not the lab director, not a committee, but one of the rank-and-file workers wanted or needed it, and created it himself. Computer games fall squarely in this tradition. Why did MIT graduate students spend hundreds of their midnights in a pursuit that benefited neither their scholarly stature, their wallets, nor their sex lives? Something lured them on. And they wanted it not because it was useful, but because it was wonderful.

Spacewar

The first video game was called Spacewar. It originated in l96l at MIT, at the time when programmers first got their hands on computers with television-like displays. The computer was a PDP-1, one of the first commercial computers that used transistors instead of vacuum tubes, and it had an oscilloscope display upon which the computer could be programmed to draw pictures. The pictures were made up of short glowing lines on the circular background of the oscilloscope screen. The images on the screen could be made to move or change shape, under control of the computer.

Spacewar was a game of spaceships battling each other. The player could turn his spaceship left or right, thrust forward using his rockets, and fire "torpedoes" at his opponents. (He could also jump into "hyperspace" if the situation became hopeless.) The goal of the game was to survive, while zapping all the opposing spaceships. The ships looked like little wedges, cigars and Fourth-of-July rockets. They picked

up momentum as thrust was applied, so that the players were constantly turning and thrusting to modify their trajectories, and then turning for a shot as an enemy ship sailed past. The player controlled his turning and shooting by pressing buttons and levers on a hand-held box connected to the computer. There was a blazing star in the center of the screen, whose gravity affected the ships' trajectories. The ships could orbit the star, but were swallowed up if they touched it.

Computer graphics was in its first breath of life in the early 60's when Spacewar was invented. At the same time at MIT, Ivan Sutherland created SKETCHPAD, a system that let the user manipulate parts of a graphical simulation with a light-pen. This opened the eyes of the "serious " members of the research community to the potential of "graphical man-machine communication." The frivolous Spacewar contained the essential elements which characterize video games today: real-time action of moving computer-generated images, interactive control by the player, and simulation embedded in a fantasy scenario.

Atari

Nolan Bushnell was an electrical engineer who had worked in a carnival as a kid. He worked for Ampex, a manufacturer of broadcast video equipment. His wacky idea about a game on a video screen, as it has often seemed to happen in Silicon Valley, didn't strike the fancy of his employers. So, in 1971, Bushnell and a few other engineers set out on their own.

They made a self-contained coin-operated version of *Spacewar* which could stand in the corner of a bar or pinball arcade. It was called While *Computer Space.* While not a total flop, *Computer Space* was not a great success either. Perhaps it was too complicated, too hard to master. The context of the man off the street investing a few quarters in a funny-looking cross between a TV set and a pinball machine was different from the context of computer technoids enjoying the fruits of their labor for free.

To get a young engineer named Al Alcorn started, just to get him up to speed, Bushnell suggested a simple exercise: make a blip of light that bounced back and forth between two movable line segments, sort of like ping-pong. Alcorn wired up some integrated circuits into a prototype, it worked, and it was fun. They decided to give it a try in a bar. Not too long after they had set up the machine, the bar owner called: their machine was out of order. When they arrived, the problem was easy to find. The coin box was jammed full of quarters.

This game was *Pong.,* which became the first video game "hit." The company, which was then called Syzygy, sold thousands of coin-operated , arcade-style Pongs for around $l500 each. A *Pong* craze swept through the bars around the country.

The time was the early 1970's, and designing computer games was somehow silly or anti-establishment enough to mesh well with the counter-culture. Atari was an anarchy. The company founded on a wacky idea explored a zillion varieties of wackiness. Anarchy may not be a very workable model for business in general, but in this case it worked because there was no need to motivate workers who were infatuated with designing games and amazed that the were being *paid* for it. It wasn't necessary (and wouldn't have worked) to schedule their creativity. Atari was powered by labors of love from its young engineers.

A flow of new arcade video games issued forth from Atari, and each new hit seemed to come from a different set of designers. *Breakout* was a one-player variation of *Pong*, in which the ball bounced back and forth between the player's "paddle" and a row of bricks, which were knocked out one by one. There were tank combat games, racecar driving games, and, yes, spaceship games. *Asteroids* was very similar to the original *Spacewar* in some ways. The player controlled a turning, thrusting, missile-firing spaceship, as in *Spacewar,* but *Asteroids* was a one-player game. The player dodged and blew

up tumbling space rocks, and occasionally dueled with a computer-controlled flying saucer. After some experimentation with more than one player, coin-operated arcade games settled into a one-player-at-a-time style along the lines of pinball machines, where several players could compete for high score, playing alternately.

The pinball business was hit hard by the competition from Atari's coin-operated video games. Some of the pinball companies, such as Bally and Williams, began making video games. However, Atari's real challenge came from Japan when a game called *Space Invaders* appeared . The player moved back and forth underneath some barriers, firing his laser cannon at an array of aliens who moved from side to side and slowly descended toward him. *Space Invaders* had some interesting innovations. The fewer aliens that remained, the faster they moved. There was a wonderful feeling of suspense as the last remaining alien raced downward, with the player needing only one good shot to end this wave of the invasion. The suspense was heightened by deep, thriller-movie music that sped up in time with the action on the screen: BOOM, boom, boom, boom, BOOM, boom, boom, boom, ...

A wildly successful arcade video game had sales in the tens of thousands, but a much larger market was waiting to be tapped, where sales were counted in the millions. In l977, Atari began selling the Video Computer System (later known as the Atari 2600), a home video game that hooked up to a television set and sold for $200. For another $20, the owner could turn it into a different video game by inserting a new game program cartridge into the black plastic console. In fact, the Atari 2600 owner could build a library of video game cartridges, costing $20 or $30 each, just as a stereo owner would build his record collection. There were 80 million households in the United States, with television sets making a 98% "penetration." (This is marketing talk for "Everybody's got a TV.") Thus there were 80 million potential sites for Atari home video games, and as this potential began to be realized, Atari's revenues rose into the billions of dollars.

The prototype for the Atari 2600 was designed by Steve Mayer and Ron Milner, two engineers who had left Ampex along with Bushnell. In contrast to earlier "dedicated" home video games which played only a single type of game, the 2600 was to be "programmable." The various cartridges programmed it for different games. The electronic hardware of the 2600 was designed to be flexible enough to play "Pong, Tank, and maybe a few other games." To keep the hardware simple and cheap, software controlled many display tasks normally done by hardware. This made writing the software for a video game difficult, but offered great flexibility, since each video game cartridge contained its own display software. The meager display resources of the 2600 (a few movable shapes and a coarse blockish background) were adapted, with clever software, to many games of very different appearances. The 2600 did offer the game designer an excellent palette of colors to choose from, with many vivid hues and shades. The production model of the 2600 was engineered by Jay Miner, an experienced chip designer, and Joe DeCuir. Putting the already minimized video game hardware into a custom-designed integrated circuit made the electronics in the 2600 very cheap indeed.

The home video games imitated arcade hits: pong games, tank games, sports games, space games. But the economics of the situation were different: a game in an arcade required a steady flow of quarters to justify its continued presence there, whereas the home video game was paid for up front. This accounts for the standard arcade mode of player-against-machine. Two equally inept humans could battle indefinitely without victory on either side, but the inexorable computer opponent, programmed to get tougher as it gets hungrier for quarters, can make sure that the action is intense and short. The home video game, on the other hand, was a fine environment for two-player games and for strategy games which required the player to sit and think for long stretches. Chess, checkers, backgammon, blackjack and bridge all appeared in the home game format, but never in arcades. The home game pace was also better for puzzling through adventure games, and for allowing time to absorb a new concept in a learning game.

Atari's 2600 had not been an instant success. Other contenders in the race for the home video game market had come from Bally, Fairchild, Magnavox, Mattel and others. The 2600 won the day because of its cheap, flexible hardware design, the variety of its game cartridges, and millions of advertising dollars. Engineers and marketeers disagree over whether design or advertising contributed

most to the success. Atari, Inc., had been sold, and the advertising money came from its new parent company, Warner Communications.

Meanwhile, in the world of coin-operated arcade games, some innovative Japanese games had appeared. *Donkey Kong* set a new standard for cartoon-like, color animation with its barrel-rolling gorilla and barrel-jumping hero, who (under player control) climbed upward to rescue the helpless blond heroine. *Crazy Climber* had more cartoonish animation, with the player climbing up the side of a building, dodging falling flower pots, and trying to stay out from under large ugly birds. It also had speech synthesis. ("Go for it! ... Oh nooooooooo...") *Pacman* became extraordinarily popular. The player steered a little gobbling face through a maze, munching dots, avoiding four goblins, and occasionally getting to chase them.

The new art of video game design evolved. *Pong*, the great hit of the early 1970's, seemed absurdly trivial to a child of the 1980's. Video game graphics progressed from black and white to color, and from low to higher resolution. The actors in these miniature dramas became more like cartoon characters than the earlier simple abstract shapes. And not only did the characters move, they were animated: walking, jumping and devouring. The complexity of the games increased, too, using more graphic objects on the screen. When the single screen began to be bloated with too many objects, the playing area exploded into a larger space, with the player's view scrolling or shifting through a number of screens.

There was also a progression in the machines on which the video games were played, although in a sense it had come full circle. *Spacewar*, the first video game, was played on the display screen of a general-purpose minicomputer. The arcade video game machines were dedicated to specific games; each game had a unique design for its electronic hardware, the buttons and levers on the front, and the artwork on the case. The home video game standardized the electronic hardware and hand controllers. Variation between games came from the different programs in the various game cartridges. When video games appeared on personal computers, such as the Apple II, the video game had returned to its birthplace, the computer, like a salmon swimming back upstream to its spawning ground.

What Is a Video Game?

A video game is an imaginary world: its inhabitants are nonexistent creatures that nevertheless the eye can see, and the hand can move. It is imaginary in the sense that there is no solid reality behind the picture. A bouncing ball may be faithfully simulated, but that moving blip of light has no real mass or elasticity. The ball's position, velocity, mass and elasticity are just numbers stored in the computer that controls the video game; and the laws of physics that govern the ball's trajectory and its bounce are just mathematical equations stored in the computer's program.

A video game usually *mimics* some real-life situation: rockets accelerating and moving in space, bouncing Ping-Pong balls, a kayak in river currents, the food-chain in an ecology. The game of chess is an abstraction based on a battle between two small groups of warriors: similarly, video games imitate life. A video game is a simulation, a model, a metaphor.

Since physical laws must be *simulated* in constructing a game's reality, the laws can just as easily be violated, and something else simulated. The impossible becomes possible. This is why video games are such a wonderful medium for fantasy. Animated cartoons have the same freedom of being beyond the laws of physics. Bugs Bunny illustrated this with his ability to paint from a can of polka-dotted paint. In another cartoon, Bugs walked around a tiny tent standing in the vast, empty desert; but when he entered it, the space inside the tent was huge. Both of these things can happen in a video game.

A video game is still a game, and it needs to have a goal, competition, and an outcome of winning or losing. Stacking up crates of cheese is an activity which could be simulated, but it is not very interesting without some mice or earthquakes to make success uncertain. There is a quality of a game called "playability," which means being challenging, but winnable. A balancing goes on here. A good game idea must usually be "tuned" to be playable by adjusting the scoring, delays, and other quantities in the game program.

A video game is made up of electronic hardware. A box full of printed circuit boards is connected to a TV-like display, and some buttons and levers for the player to push. There is a computer in the box of circuitry, and the program that defines the game is in the computer's memory. (See the block diagram of Figure l-l.)

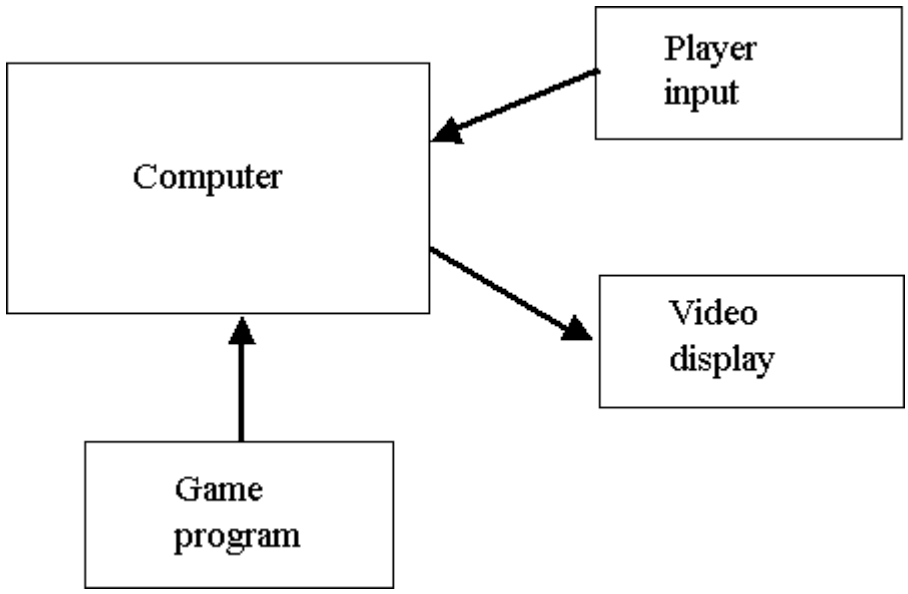

Figure 1-1
Block Diagram of a Video Game

The program is the heart of a video game. It defines the shapes, their colors, how they move, and how they respond to the player's inputs. The program is a list of instructions to the computer about what to do in different situations. (If the ball touches the wall, then bounce.) The video game designer creates a program that defines his or her game; the program enforces the rules of the game and moves the little shapes around as the player plays it. The designer is a legislator, the program is the law he invents, and the computer is the cop who makes sure the law is obeyed.

Video game design is a new art form, rich with unexplored possibilities. Most forms of art have their technical problems, so it is not so strange that a video game designer needs to be a good programmer. An artist making kinetic sculptures needs to be part mechanical engineer. The game designer needs skill in both conception and execution. He needs to think about what kind of ideas make good video games and know how to turn a good idea into a working program.

Chapter 2

The First Text-Based Adventure Game: *Colossal Cave*

Willie Crowther liked to crawl around in caves. Periodically, he went to Kentucky to join other amateur spelunkers in exploring Bedquilt Cave, a huge cavern connected to Mammoth Cave. Crowther used the computers where he worked, at Bolt Beranek and Newman (BBN) in the Boston area, to store topographic data and produce maps of the cave. In l975, when he couldn't go anymore to the caves in Kentucky, he used Bedquilt Cave as the setting for a spelunking simulation, in which he could move from chamber to chamber in the cave by typing commands to the computer. In response to his movement commands, the program printed a textual description of each chamber as he entered it. He could also carry "objects" from chamber to chamber in the cave, pick up new objects found along the way, and drop carried objects so that they would be left behind when he moved to a new chamber. Crowther made the program into a game by defining some objects as treasures, and putting these treasures behind obstacles like chasms and giant snakes. The player needed to find and use other objects in the cave to get past the obstacles and claim the treasures. The game had about a dozen objects and 40 rooms. He called it *Adventures*..

The design of *Adventures* was influenced by *Dungeons and Dragons* , a non-computer fantasy role-playing game which Crowther had been involved in playing. In *Dungeons and Dragons*, a small party of characters explored an unknown subterranean labyrinth full of treasurers and monsters. Each character, who might be a magic-user, fighter, cleric, elf, or thief, was controlled by a human player. A character had a name, and attributes of strength, intelligence, wisdom, constitution, dexterity, and charisma. A character carried along with him supplies, weapons, armor, magic potions, and other useful objects. The party moved through the dungeon, mapping it, searching for secret doors, examining (possibly booby-trapped) treasure chests, and battling monsters. The unknown layout of the dungeon, and the locations of various perils and windfalls, were presided over by a person called the "Dungeon Master." Game play was a dialogue between the exploring players and the Dungeon Master. The Dungeon Master told them about their immediate surroundings and the results of their actions. He used a map of the dungeon which was kept hidden from the players. *Adventures* was *Dungeons and Dragons* for one player (a "solo dungeon") in which the computer was the Dungeon Master.

Crowther let the player direct the game with English nouns and verbs. He chose this natural language input, rather than a complicated, formalized command language, because he wanted to make a game that would not intimidate non-computer people. He also designed the game for his kids.

Crowther made the program available to be played on the ARPAnet, one of the first computer networks. The ARPAnet linked university computer science departments and other research institutions across the United States. Don Woods, at the Stanford Artificial Intelligence Lab in California, played the game over the ARPAnet. Woods was intrigued. To get the program's source code, he tracked down

Crowther (who had by then left BBN) by sending a message to every site on the ARPAnet. He located Crowther, got a copy of the program, and began playing with it.

Don Woods greatly expanded the game, adding many new rooms, objects, commands and treasures. Whereas Crowther's network of chambers really did describe Bedquilt Cave in Kentucky. Wood's additions were completely fanciful, and owed more to Tolkien's *Lord of the Rings* trilogy than anything else. (The game's name lost its final "s" somewhere along the way to become, simply, *Adventure*.). The graduate students working with Woods at the AI Lab followed the development of *Adventure* with avid interest. They tested their wits against the game's puzzles. As the game developed, feedback from this community of players let Woods make sure that his puzzles were solvable, and let him provide humorous responses for often-tried commands. Wood's roommate, Bob Pariseau, who was, like Woods, also involved in the computer industry, helped with the development of the game. As had been the case with*Spacewar* at MIT, the young graduate students responded with instinctive fascination to *Adventure*. Many of them explored the space of rooms and objects within the game, and a few were inspired to explore the very idea of an adventure game -- that is, to write their own games. The responses of the computer science professors working at the AI Lab ranged from mild interest to outrage at the diversion of resources to such a frivolous end.

Woods made the game accessible on the ARPAnet so that his friends at MIT could play it, and he left for two weeks vacation. On his return, he found that the computer at the AI Lab had been swamped with people connecting over the network to play the game. Playing *Adventure* was soon restricted to nights and weekends.

Crowther had written his *Adventure* program in FORTRAN, a computer language universally available on scientific and commercial computers. Wood's additions were also in FORTRAN. The program was therefore easily transportable to other computers. In 1977, Woods gave the source code to Ken Harrenstien, who got it running at the MIT Artificial Intelligence Lab. The result there was that "no one got any work done for a week." Requests for copies began pouring in. Woods wrote a few pages of notes about adapting the program to various computers, and turned it loose. *Adventure* spread rapidly across the country, first over the ARPAnet, and then to computers in hundreds of companies. Releasing the source code did have the advantage that Woods could not get in trouble at Stanford over computer time being wasted on *other people's* computers. Looking back, Woods said that releasing the source code was "done deliberately, though perhaps not with full understanding of what would ensue."

Adventure became wildly popular among computer people in 1978. Before the infatuation with that particular game had quite worn off, adventure games with new puzzles, on new themes, were springing up all over.

A Text Dialogue

Adventure was a dialogue between the player and the computer. It began like this (the player's commands are in **BOLD CAPITALS)**:

```
Welcome to Adventure!!  Would you like instructions?

YES

Somewhere nearby is Colossal Cave, where others have found
fortunes in treasure and gold, though it is rumored that
some who enter are never seen again.  Magic is said to work
in the cave.  I will be your eyes and hands.  Direct me
with commands of 1  or 2 words.  I should warn you that  I
```

```
only look at the first five letters of each word, so you'll
have to enter "northeast" as "ne" to distinguish it from
"north".  (Should you get stuck, type "help" for some
general hints.  For information on how to end your
adventure,  etc., type "info".)
                                                    ---
This program was originally developed by Willie
Crowther.  Most of the features of the current program were
added by Don Woods (DON @  SU-AI). Contact Don if you have
any questions, comments, etc.

You are standing at the end of a road before a small brick
building.  Around you is a forest.  A small stream flows
out of the building and down a gully.
```

At this point the program paused, waiting for a command from the player. If the player typed the command

ENTER BUILDING

then the program responded

```
You are inside a building, a well house for a large spring.

There are some keys on the ground here.

There is a shiny brass lamp nearby.

There is food here.

There is a bottle of water here.
```

The keys, lamp, food and water were objects which could be picked up and carried to other places.

TAKE KEYS
OK

TAKE LAMP
OK

The objects which the player picked up stayed with him as he moved.

LEAVE BUILDING
You're in front of building.

INVENTORY
You are currently holding the following:
Set of keys
Brass lantern

The player often used compass directions to move from place to place.

GO SOUTH
You are in a valley in the forest beside a stream tumbling
along a rocky bed.

GO SOUTH

```
At your feet, all the water splashes into a 2-inch slit in
the rock.  Downstream the streambed is bare rock.
GO SOUTH
You are in a 20-foot depression floored with bare
dirt.  Set into the dirt is a strong steel grate mounted in
concrete.  A dry streambed leads into the depression.

The grate is locked.

GO DOWN
You can't go through a locked steel grate!

You're outside grate.

The grate is locked.
```

The locked grate was an obstacle that prevented the player from getting beyond it. The player was confronted with a puzzle here. There was a way of getting past the locked grate by performing some particular action, sometimes using a particular object. The problem was to figure out which action, and which object. The player might try

```
BREAK GRATE
It is beyond your power to do that.
```

or perhaps

```
PRY GRATE
Sorry, I don't know the word "PRY".
```

Asking for help, as suggested in the game's initial instructions, defined the rules of this guessing game a little bit better.

```
HELP
I know of places, actions, and things.  Most of my
vocabulary describes places and is used to move you
there.  To move, try words like forest, building,
downstream, enter, east, west, north, south, up or down.  I
know about a few special objects, like a black rod hidden
in the cave. These objects can be manipulated using some of
the action words that I know.  Usually you will need to
give both the object and action words (in either order),
but sometimes I can  infer the object from the verb
alone.  Some objects also imply verbs;  in particular,
"inventory" implies "take inventory", which causes me to
give you a list of what you're carrying.  Some objects have
unexpected effects; the effects are not always
desirable!  Usually people having trouble moving just need
to try a few more words.  Usually people trying
unsuccessfully to manipulate an object are attempting
something beyond their (or my!) capabilities and should try
a completely different tack.  One point which is often
confusing to beginners is that, when there are several ways
to go in a certain direction (e.g., if there are several
holes in the wall), choosing that direction in effect
chooses one of the ways at random; often, though, by
specifying the place you want to reach you can guarantee
choosing the right path.  Also, to speed the game you can
sometimes move long distances with a single word.  For
```

```
example, "building" usually gets you to the building from
anywhere above ground except when lost in the
forest.  Also, note that cave passages and forest paths
turn a lot, so leaving one place heading north doesn't
guarantee entering the next from the south.
Good luck!
```

The obstacles in *Adventure* were logical puzzles. The nature of an obstacle usually provided a clue to the means of getting past it. For example, faced with a locked grate, the player often tried

```
UNLOCK GRATE
You have no keys!
```

This response suggests that it might be worthwhile to go back and get the set of keys dropped outside the building.

```
GO BUILDING
You're in front of building.

There are some keys on the ground here.

TAKE KEYS
OK

GO GRATE
You're outside grate.

The grate is locked.

UNLOCK GRATE
The grate is now unlocked.

GO DOWN
You are in a small chamber beneath a 3x3 steel grate to the
surface.  A low crawl over cobbles leads inward to the
west.

The grate is open.
```

The solution to the locked grate obstacle was twofold: to have the keys-object and to give the command "UNLOCK." This was typical of the puzzles in *Adventure*. The non-treasure objects in the game all functioned as tools to get past obstacles. Some of the treasures, too, served as tools.

Going further into the cave turned up some interesting objects.

```
GO WEST
You are crawling over cobbles in a low passage.  There is a
dim light at the east end of the passage.

There is a small wicker cage discarded nearby.

GO WEST
It is now pitch dark.  If you proceed you will likely fall
into a pit.
```

This darkness was another obstacle, another puzzle. Solving it revealed that

```
You are in a debris room filled with stuff washed in from
the surface.  A low wide passage with cobbles becomes
plugged with mud and debris here, but an awkward canyon
leads upward and west.  A note on the wall says "MAGIC WORD
XYZZY".

A three-foot black rod with a rusty star on an end lies
nearby.

GO WEST
You are in a splendid chamber thirty feet high.  The walls
are frozen rivers of orange stone.  An awkward canyon and a
good passage exit from east and west sides of the chamber.

A cheerful little bird is sitting here singing.
```

There were indeed uses for the wicker cage, the rod with the star on the end, the magic word, and the cheerful little bird. Deeper in the cave were fifteen treasures, hidden within a network of more than a hundred chambers. The player's goal was to gather treasure, and to do that, he needed to explore and map the cave, and experiment with the various objects in order to discover their functions. The game gave the player a powerful impression that he was exploring an unknown, mysterious world.

Direction Verbs, Action Verbs, and Nouns

The structure of the program for Adventure was hinted at when it told the player

```
I know of places, actions, and things.
```

The program recognized three types of words: direction verbs, action verbs, and nouns. Direction verbs were used to move through the network of places that comprised the world of Adventure. Both place names (like "BUILDING" or "GRATE") and true direction words (like "EAST" or "UP") were considered to be direction verbs, along with some intermediate cases (like "ENTER" or "DOWNSTREAM"). Each place had some descriptive text associated with it which was printed when the player moved there.

```
You are at one end of a vast hall stretching forward out of
sight to the west.  There are openings to either
side.  Nearby, a wide stone staircase leads downward.  The
hall is filled with wisps of white mist swaying to and fro
almost as if alive.  A cold wind blows up the
staircase.  There is a passage at the top of a dome behind
you.
```

For brevity, places also had short descriptions which were used when a player had already visited a place before.

```
You're in Hall of Mists.
```

Nouns corresponded to objects. The defining characteristic of objects was that they could be carried from place to place. Thus each object had a location. Besides location, each object had a state, which affected the description of the object. For example,

```
There is a shiny brass lamp nearby.
```

described the brass lantern, which, however, when it had been lit, looked like this:

```
There is a lamp shining nearby.
```

When an object was being carried, it was given a very brief description in the inventory list

```
Brass lantern
```

which did not distinguish among its various states.

Action verbs changed the states of objects ("LIGHT LAMP"), handled picking up and dropping objects, and performed other functions. Whereas the places and objects in the Adventure program were defined by tables giving their descriptions and other attributes, action verbs were all handled by special purpose code. (The program consisted of a main program of about 2000 lines of FORTRAN, another l000 lines of subroutines, and 2000 lines for the data base that defined the places and objects.) Most of the action verbs worked in conjunction with particular objects, and had the purpose of getting past particular obstacles.

Active Creatures

Many menacing creatures were encountered in playing Adventure, but most of them were passive, only blocking the player's progress, and could be better thought of as obstacles.

```
A huge green fierce snake bars the way!
```

However, there were two types of creature that moved around, actively harassing the player: dwarf and pirate. A creature was best considered to be a type of object which moved around on its own, initiating actions. Dwarves were prone to show up at unpredictable times, and were rather unfriendly.

```
There is a threatening little dwarf in the room with you!

One sharp nasty knife is thrown at you!

It misses!
```

In Crowther's version of Adventure, dwarves followed a fixed path through the cave; Woods let the dwarves do a random walk, so that they could end up in any part of the cave. However, this introduced a problem: there were a couple of mazes ("of twisty little passages") in the cave which were easy to get into, and hard to get out of by random wandering. These mazes tended to act as dwarf-traps, so that if the unfortunate player ever did blunder into one, he was quickly assaulted by several dwarves. Woods therefore fixed the dwarves' wandering procedure to avoid the mazes unless the player was in one of them.

The pirate wandered around aimlessly, like the dwarves. (There were several dwarves, but only one pirate.) When the pirate ran into the player, he stole treasure from the player.

```
Out of the shadows behind you pounces a bearded pirate!
"Har, Har," he chortles, "I'll just take all this booty and
hide it away with me chest deep in the maze!"  He snatches
your treasure and vanishes into the gloom.
```

The pirate was implemented in the program as a dwarf with special behavior.

Gathering Treasure

The goal in Adventure was to gather treasure, thereby scoring points.

```
To see how well you're doing say "score".  To get full
credit for a treasure, you must have left it safely in the
building, though you get partial credit for just locating
it.

SCORE
If you were to quit now, you would score 59 out of a
possible 430.
```

The scoring system clearly advertised when undiscovered treasures remained and served as a yardstick against which the player could measure his progress.

Most of the game's puzzles had a treasure lying just beyond.

```
There is a large sparkling nugget of gold here!
```

Gathering treasure was a consistent fantasy that united the diverse puzzles of Adventure. Solving the puzzles was the real challenge of the game, but there was something satisfying about going and grabbing the treasure after an obstacle had been overcome.

Characteristics of Colossal Cave

Adventure was a game of exploration and problem-solving in a fantasy world. To explore is to travel into unknown regions, and so the map of the fantasy world functioned as a space through which the player could explore. It is important that the game world could not be seen all at once. If this had been so, moving through the world would expose no new information, hence there would be no exploration.

Exploration was encouraged in Adventure. A player nearly always drew a map to help navigate through the cave. Besides exploring the space of places, the player also explored the space of verbs, seeking to discover what various objects were good for. (It could be frustrating. Adventure was called a "guess-my-command game" by some.)

Adventure was an elaborate logical puzzle. The active dwarves and pirate enlivened the game and avoided total predictability, but the bulk of the game was still passive, waiting to be figured out. It was really more of a puzzle than a game, since once a player solved it, he was done. Asked if he had been part novelist when he created Adventure, Woods replied, "A novelist -- no; an artist -- perhaps; a logician -- certainly."

The "plot" of Adventure depended on choices made by the player. (It might be more accurate to say that the player discovered the true plot -- the solutions to the puzzles -- in the course of playing the game.) A truly interactive narrative like this had never been seen before. The novelty of this interactiveness made it easy to overlook the more ordinary merits of the narrative itself. The writing was good in Adventure. The use of language was precise and colorful. The fantasy setting of Adventure was eclectic, containing magical beings and objects, monsters, and modern machines.

Adventure was full of humor. While Woods was developing the program, people were playing it at the AI Lab, so that he got plenty of feedback about the phrases that people were likely to try in various situations. Some of these often-tried commands, even though they didn't get past any obstacles, did elicit

their own special replies from the program, so that the program seemed to comment upon the player's attempted solutions with a friendly sarcasm.

```
OPEN SESAME
Good try, but that is an old worn-out magic word.
```

Or, in another situation

```
There are a few recent issues of "Spelunker Today" magazine
here.

READ MAGAZINE
I'm afraid the magazine is written in dwarvish.
```

Wood's Ph.D. dissertation in computer science was "Traversing Planar Graphs." This topic fell into graph theory, an accepted branch of computer science. In developing Adventure, Woods was helping to create a new genre of computer games, and, in fact, a new form of art. It is ironic that in the eyes of most computer science professors at the time, computer games lacked the "seriousness" they deemed necessary for true research. This is a commentary on the academic world, and its relation to innovation and creativity. It permits "serious" creativity. Woods probably could not have written his thesis on "Getting Past Huge Green Fierce Snakes, and Other Logical Puzzles."

Adventure could stand alone on its own popularity, but its greatest significance was in establishing a new genre of computer games. The very existence of Adventure made it obvious, as it must have been with the first novel, that many more examples of this new form could be created, in different settings, with different characters. Inevitably, it was imitated, and a whole legion of adventure games appeared. The first ones inherited the text dialogue style of the original Adventure, and later mutations embraced graphics and animation.

Chapter 3

Adventure As a Video Game: *Adventure* for the Atari 2600

I had played the new sensation, the original Adventure, when I finished designing my first video game. The time was June l978, I worked for Atari, and my next order of business was to begin working on another game cartridge for the Atari 2600 home video game. I had a scheme for adapting the text dialogue of Adventure into a video game -- use the joystick to move around, show one room at a time on the video screen, and show objects in the room as little shapes. I hoped the program to do this might somehow fit into the tiny (4K byte) memory available in a game cartridge, and so, in spite of my boss's skepticism, my infatuation with Adventure swept me into a mad frenzy of programming.

A month later, I had a prototype: the player could move a small square "cursor" from screen to screen, and could pick up the little colored shapes that were found on some of the screens. The screens were connected edge-to-edge. And there was a bothersome dragon that chased the cursor around, trying to eat it. Exhausted, I went on vacation, and found, on my return, that Atari upper management had decided that I should turn my fledgling adventure game into a video game about Superman. Atari's parent corporation, Warner Communication, owned the soon-to-be-released Superman movie, and a Superman video game could ride on the wave of "hype." I squirmed, and soon wriggled out of that assignment: my coworker John Dunn agreed to take over the program and turn it into Superman leaving me free to develop the same program in a different direction, namely, to continue with Adventure. (This sort of contrariness later caused Atari executives to label their video game designers "a bunch of high-strung prima donnas.") I moved forward, discovering how ideas from the text adventure game could be made to work as moving shapes in a video game, and discovering what the young tradition of video games could contribute to the new genre of adventure games. The program took eight months, from start to finish; Atari marketed the cartridge, and since Woods's game was in the public domain, the video game, too, could be called Adventure.

This new video version of Adventure was a quest: the player started out beside the Yellow Castle with the goal of retrieving the Enchanted Chalice, which was out there somewhere in the network of thirty rooms. (See Figure 3-1). To make things difficult, three dragons infested the game, chasing the player from room to room, and trying to eat him. A giant bat also causes trouble, moving objects around and stealing things from the player. There were a number of useful objects. The sword killed dragons. The bridge let the player cross walls in the maze. The magnet sucked out objects which were stuck in the walls. The black, white and yellow keys each unlocked a castle of matching color. (See Figure 3-2).

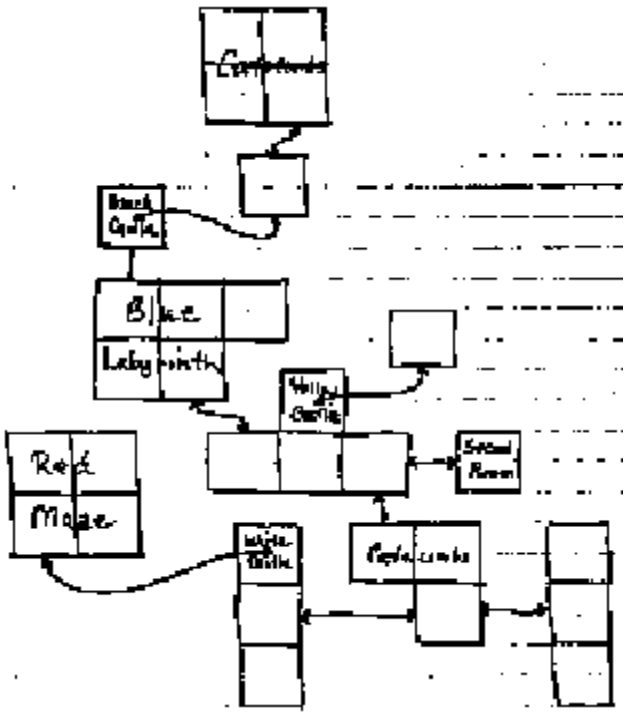

Figure 3-1
Map of the Network of Rooms
in Atari 2600 Adventure

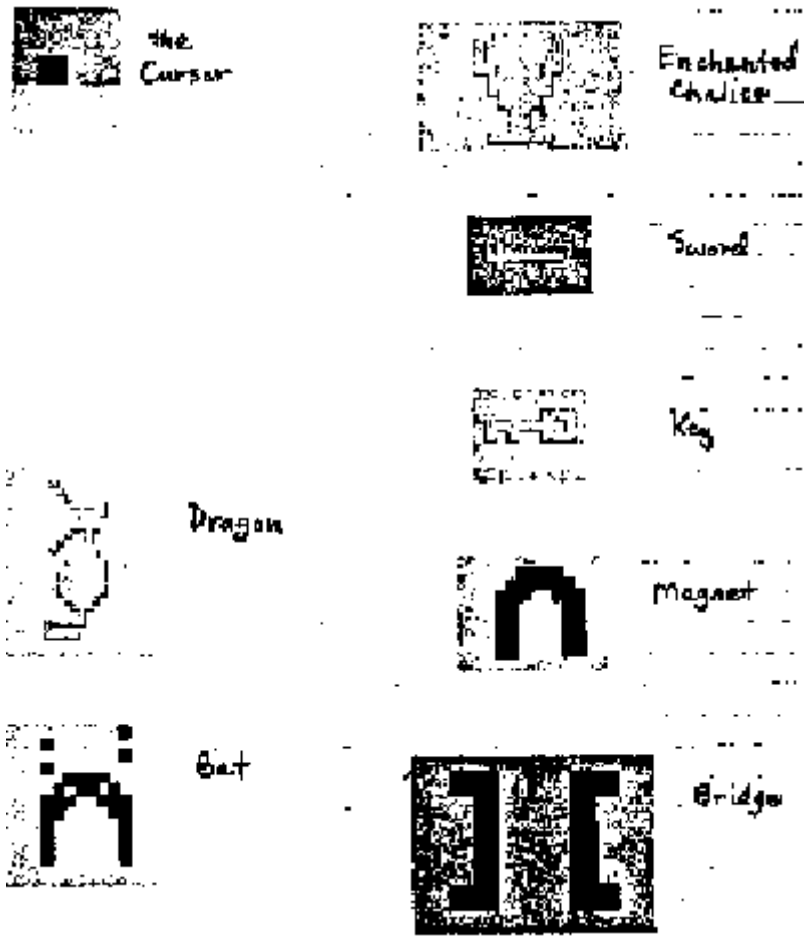

Figure 3-2
Creatures and Objects
in Atari 2600 Adventure

Each of the three castles in Adventure had interiors composed of one or more rooms. The player entered a castle by moving into the portcullis grillwork in its doorway. However, the portcullis could be raised or lowered under control of a key, and when the portcullis was down, the castle was locked. Entering or leaving a locked castle was not possible. Each of the three castles had its own key object, and all the castles were locked when the game began.

This video game, Atari 2600 Adventure, was inspired directly by Crowther and Woods' text Adventure. I tried at first to create video game counterparts of features in the text game. The magic rod can create a crystal bridge to span an impassable fissure in the text version; I tried a rod-shape which, when it touched a maze wall, caused a bridge-shape to appear. The "maze of twisty little passages, all alike" became a very confusing 8-room video maze. These direct transliterations from text to video format didn't work out very well. While the general idea of a video game with rooms and objects seemed to be a good one, the graphic language of the video game had different strengths than the verbal language of the text dialogue. Just as the art form of film slowly diverged from its parent, drama, the animated adventure game diverged from the text adventure game because of the difference between the medium of text and the medium of animated graphics.

Player Input

The adaptation of the adventure game to the video game medium required a radical change in the form of the player's commands to the game. Typing was not possible on the Atari 2600 video game; the standard input was a joystick with one "fire" button on it. The video game player could push the joystick lever in one of four directions, or press the button. The text player, on the other hand, typed in a two-word command, composed of an action verb and a noun. There were dozens of words in the text game's vocabulary, both for nouns and for verbs; in two-word combinations, there were thousands of possible commands. How could the video game player initiate the wide variety of actions that were possible in the text game?

The joystick was a natural for north-south-east-west movement. There is something satisfyingly responsive about shoving the joystick lever and thereby moving a shape on the screen in the same direction. It is important that the player can hold onto the single lever and move it in any of several different directions. The lever itself fades out of consciousness, and the player feels that he is propelling the cursor with his own muscles, as if he were scooting a brick around on a sidewalk.

The player identifies himself with the shape he moves around on the video screen. When a player says "I ran into a wall." he means the shape he moved ran into a wall; he is that shape. In Atari 2600 Adventure, this self-shape is a little solid-colored square. It can be called a cursor, since its function as a position indicator is similar to the rectangular blinking cursor found on screens of text. I originally called it "the man."

Besides movement, picking up and dropping objects are the most important player actions in an adventure game. With the joystick lever assigned to movement, the single button on the base of the Atari joystick was the clear candidate for grappling with objects. It was not obvious exactly how the button should control taking and dropping objects. If the function of "take," for instance, was to be invoked when the button was pressed, which object should be taken? The graphical nature of the video game provided a solution to this. In the world of video games, objects on the screen usually interact only when they run into each other. This is called a "collision," and is defined as an overlap of one shape with another. The object-shape to be taken could be specified by moving the cursor to touch it. In fact, the collision itself could invoke the pick-it-up action; this left the button free for dropping. But drop what? If several objects had been picked up, a selection was again needed. And how should the carried objects be shown? Pushing the button could have called up an "inventory" screen, and the cursor could have been steered into the object to be dropped. However, a simpler solution was adopted: only one object could be carried at a time, and it was shown besides the cursor on the screen. Pushing the button dropped it.

This approach had several advantages. It was simpler, so the program was shorter. The screen always showed the room the player was in, and what he carried; the player didn't have to worry about being eaten by a dragon that came by while he was examining his inventory. Of course, time could have been suspended during inventory-view, but that didn't seem right for a real-time game. The limitation of being able to carry only one object gave the player some interesting strategic choices: which object should he carry -- the treasure or the weapon?

Since the object being carried was shown on the screen, it had a position relative to the cursor. The player could adjust the positioning of his held object. When exploring unknown dragon-infested territory, it usually made sense to have your sword out front, because simply holding a sword did not prevent a dragon from eating you -- but poking the sword into the dragon did. For a dragon which was scared of the sword, it worked better to loiter near a room boundary, with a behind-the-back sword dangling into the next room, ready to make a swift stroke when the dragon came into striking range.

What about all the other actions that a player might want to initiate in an adventure game? These, too, could be specified by touching objects together. For example, in a text game, one might command "KILL

DRAGON." The corresponding action in the video game was to pick up a sword and touch it to the dragon. In a sense, the held object and the object touched were the analogs of the action verb and noun from the text adventure.

A great variety of "commands" might be given if the player had the right verb-objects at hand. Placing the bridge object across a maze wall and going across it was equivalent to "CROSS WALL." Touching a key to a castle's portcullis commanded "unlock castle." Bringing the magnet into a room to retrieve a sword stuck in the wall was like "ATTRACT SWORD." Thus, the syntax of nouns and verbs in the text adventure had an analog in a video adventure -- a "syntax" of overlapping shapes.

Objects

Objects which could be picked up, carried from place to place, and used for various tasks were described with a phrase of text in the original Adventure:

A three-foot black rod with a rusty star on the end lies nearby.

Such an object became, in the graphic language of the video adventure game, a little colored shape that appeared at some location on the screen. Each object had a location, which consisted of a room and an (X,Y) position within that room.

Each object in Atari 2600 Adventure does something. The keys open castles; the sword kills dragons; the bridge crosses walls; the magnet retrieves lost objects; and the chalice wins the game. It would be possible, of course, to have useless objects in a game, to serve as decoys or decorations, but in Adventure every object has a function. The objects are really tools, since the player can use them to cause things to happen in the game world.

Whereas the goal of the game in the original Adventure was treasure-gathering, video Adventure is defined as a quest. One single treasure, the Enchanted Chalice, must be located and brought home.
Thus, the tool-objects must contribute somehow to the overall goal of the quest. For example, if the chalice is locked inside the Black Castle, then finding the Block Key becomes a subgoal, subordinated to the primary goal of getting to the chalice . If the Black Key is found, but is inaccessible because of the dragon guarding it, then another subgoal is spawned -- find the sword so as to get past the dragon. Each tool-object is a means of getting past a certain kind of barrier. Since needed objects may be behind barriers which, in turn, require other objects, a hierarchy is created of goals and subgoals. Figure 3-3 shows the arrangements of some obstacles and their solutions.

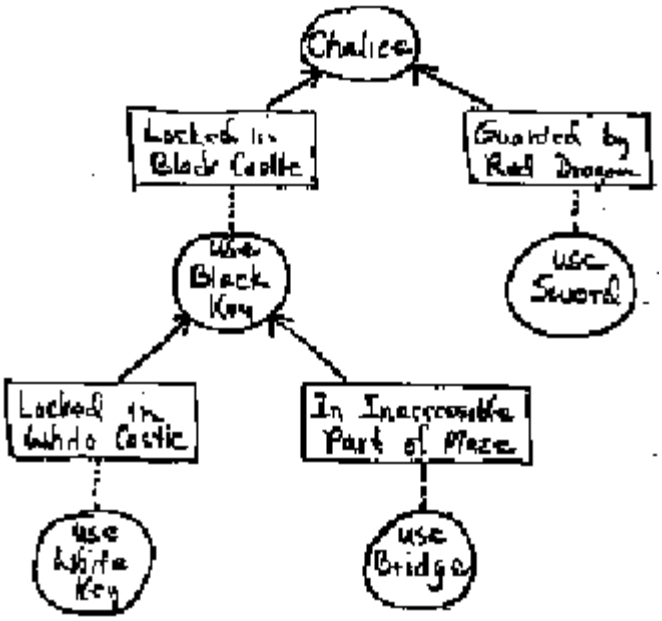

Figure 3-3
Hierarchy of Goals and Subgoals
in Atari 2600 Adventure

Creatures

A creature in an adventure game is an object that moves around on its own, initiating actions. It is best to consider a creature as a special type of object so that the creature can inherit the traits already defined for objects: shape, color, location and ability to be picked up. Each type of creature has some special rules that specifies how it behaves, what it responds to, and what actions it can take. These special rules are defined by a part of the game program, usually a subroutine, that corresponds to the creature type. There are two species of creature in Adventure: dragon and bat.

Dragons are the main villains in the game. There are three of them, and they chase the player around, trying to eat him. If the player's reflexes are too slow, or if he gets cornered, the dragon swallows him. Once eaten, the player can be "reincarnated" to get out of the dragon's belly, but as penalty he goes back to the starting location, loses whatever he was carrying, and any dragons he may have killed are reincarnated, too. (Having killed dragons is like being vulnerable in the game of bridge: there is more to lose in the battle with the third dragon. Getting eaten means that the two dead dragons will come back to life.)

There are four states that a dragon can be in, each shown on the screen by a different dragon-image: chasing the player, biting him, having swallowed him, and being dead. The state diagram of Figure 3-4 shows the conditions in the game that cause transition from one dragon-state to another. In a typical interaction between player and dragon, the dragon goes back and forth between the biting and chase states several times as it tries to eat the player, and then either the player gets away, gets eaten, or kills the dragon with the sword. The dragon graphics change rapidly, mirroring the state changes. Not only is this interesting animation, but it also gives the player valuable visual feedback about which state the dragon is in at any given moment.

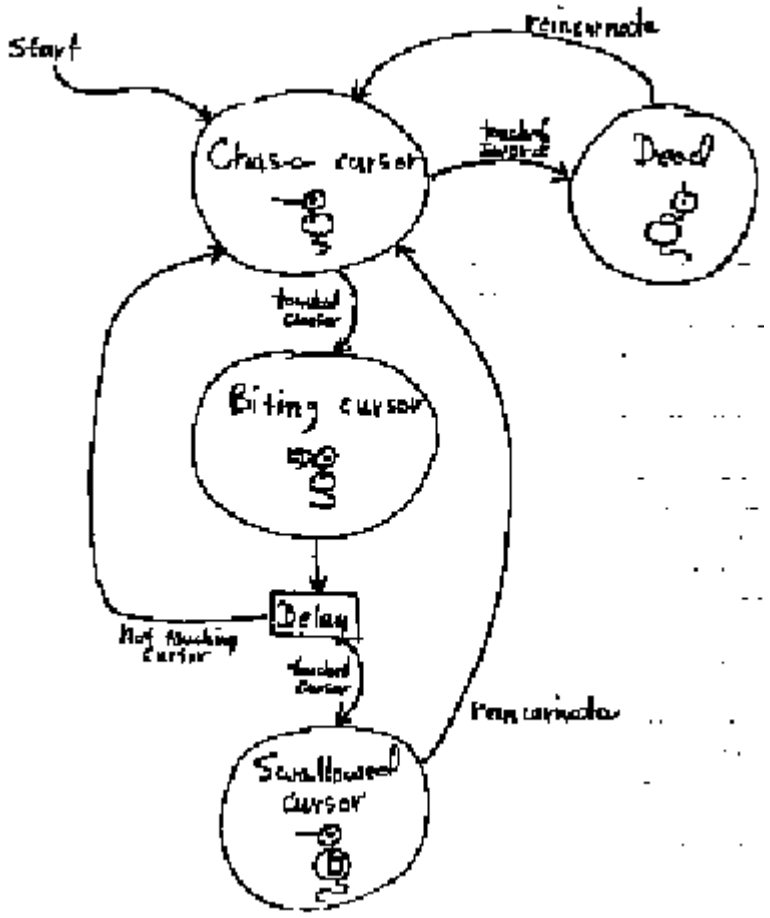

Figure 3-4
State Diagram of Dragon

For the dragon to succeed in swallowing the player's cursor, it must collide once with the cursor (entering biting state), and after a fraction of a second, collide again. This brief delay gives the player time to recoil, and if his reflexes are fast enough, then he avoids being swallowed and the dragon resumes chasing him. As the player and dragon go repeatedly through this chase-bite cycle, neither has the advantage. The dragon wins the battle if it can swallow the player; the player wins if he can get to a sword and use it to kill the dragon.

It is significant that it takes two collisions with the dragon, not one, for the cursor to be swallowed. Merely colliding with the dragon is not fatal. Thus the interesting chase-bite cycle is made possible. Most video games define simple one-time collision with enemies as fatal and irrevocable, and thereby miss a chance to create a more interesting interaction.

The length of the recoil interval between biting and swallowing is quite important. If it is too long, it is trivial to avoid being eaten, and the player can ignore the dragon and do whatever he wants. If the interval is too short, the player never succeeds in recoiling, and gets eaten every time. There is a middle ground between "trivial" and "impossible" called "challenging." Trying out the game with various players and watching how well they do is the best way to adjust a game's timing. This process is called "tuning" the game. Varying the length of the recoil interval turned out to be an effective means of varying the game's

difficulty. It ranges from around a tenth of a second at the most difficult, to about three seconds at the easiest.

The bat is the second species of creature in Adventure. The bat flaps from room to room carrying along an object, and periodically, he tires of his current trinket, and discards it in favor of a new object to carry off. Without the bat, non-creature objects would never move from the spots where the player dropped them. The effect of the bat is to move objects around, to disturb the predictability of the game. The bat is the game's confusion factor.

As detailed in Figure 3-5, the bat has two states: seeking a new object to pick up, and carrying off a newly acquired object, ignoring all other objects. This ignore-state, which lasts for about ten seconds after a new object is picked up, was needed to make sure the bat would carry its new trinket off to another room. In an early version of the game, the bat reentered seek-state immediately after picking up a new object; in a room containing two objects (plus the one carried by the bat) the bat would ferry objects back and forth between the two positions forever, never leaving the room.

The bat can carry dragons. It sometimes happens that the bat will appear carrying a dragon, steal the player's sword and fly off with it, leaving the disarmed player to deal with the left-behind dragon.

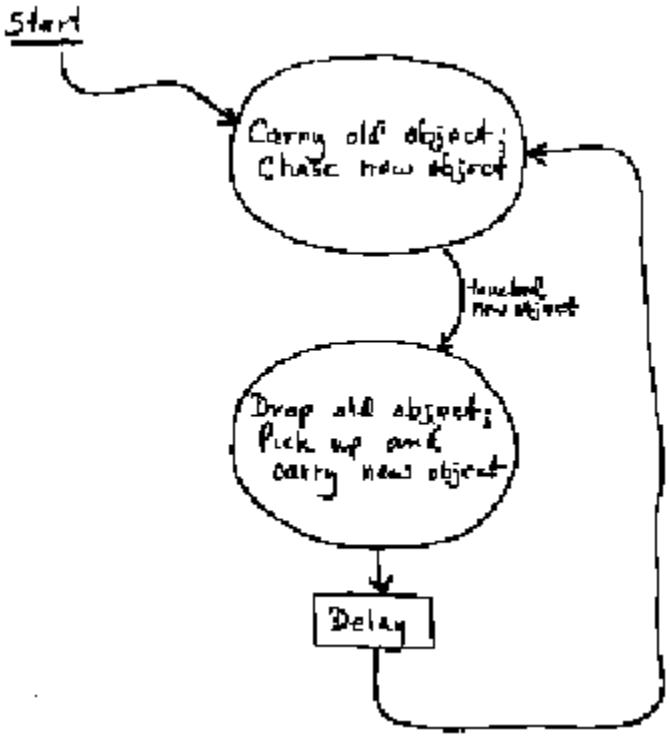

Figure 3-5
State Diagram of Bat

Mazes

In a text adventure, a room is a single location. Although there are passages to other rooms, the room itself has no internal structure. A video adventure, by comparison, allows the player to have a position within a room, shown on the screen by the cursor's position. A single room can show a simple maze on the screen, with passages going off the screen to other (as yet unseen) maze-rooms. The walls of the maze, of course, block the cursor's movement. A 4- or 5-room maze can be quite complicated.

Besides forming mazes, walls prevent the player from leaving a room in certain directions, and thus help form the overall layout of the network of rooms. In some rooms in Woods's text adventure, typing "GO EAST" would produce this response:

> There is no way to go that direction.

Walls perform the analogous function in a graphical adventure game.

A maze is a geometric construction in space; the positioning of its walls defines a maze. Video graphics do an excellent job of capturing the geometry of a maze. By contrast, using sentences to describe a maze is inefficient and piecemeal. A player lost in the maze of "twisty little passages" in Woods's text adventure invariably draws a map if he wants to get out by understanding the maze, rather than by just the luck of random wandering. The verbal representation of a maze is abstract, whereas the graphical representation is concrete and therefore less confusing. Five-year-olds learn the path to the Black Castle through the five maze rooms of the Blue Labyrinth in Adventure. Moving through the visual depiction of each room lets the child remember his path in the same way that he remembers the route from living room to bathroom in a friend's house.

One leaves a room in video Adventure by driving off the edge of the screen. Since the screen has four edges (top, bottom, left and right), every room has four links to other rooms (to the north, south, west and east). Going from one room to another in an adventure game is like using a Star Trek transporter. ("Beam me up, Scottie!") The player vanishes from one place and materializes in another place. The two places are not necessarily "near" each other: they are merely connected. It is often impossible to draw a map of an adventure game's network of rooms so that all linked rooms are side by side. The map of Adventure (Figure 3-l) is an example of that; although small groups of rooms can show links among each other by adjacency, there are leftover links that must be shown explicitly with lines running between the two connected places. If a map, which is a shrunken replica of a region's geometry, cannot be constructed, then that region cannot be built full-sized, either. In other words, these places are impossible. Adventure games simulate spaces that can't exist in the physical world. But it doesn't really matter that a map or scale model of these places cannot be built; the player can still move from room to room in the game. As in the Bugs Bunny cartoon with the vast volume inside the tiny tent, these impossible spaces have surprising properties.

A network of rooms, depending on how the rooms are interconnected, can have inconsistent geometry. One might assume that a lot of square screens connected edge-to-edge could be thought of as a big array of screens, arranged as rows and columns in a plane, just like the square tiles on a kitchen's linoleum floor. However, rooms can be interconnected in a way that is inconsistent with plane geometry, and with common-sense expectations about moving through space. Three inconsistencies that occur are non-retracible paths, non-unique diagonal rooms, and wrap-around paths. (See Figure 3-6).

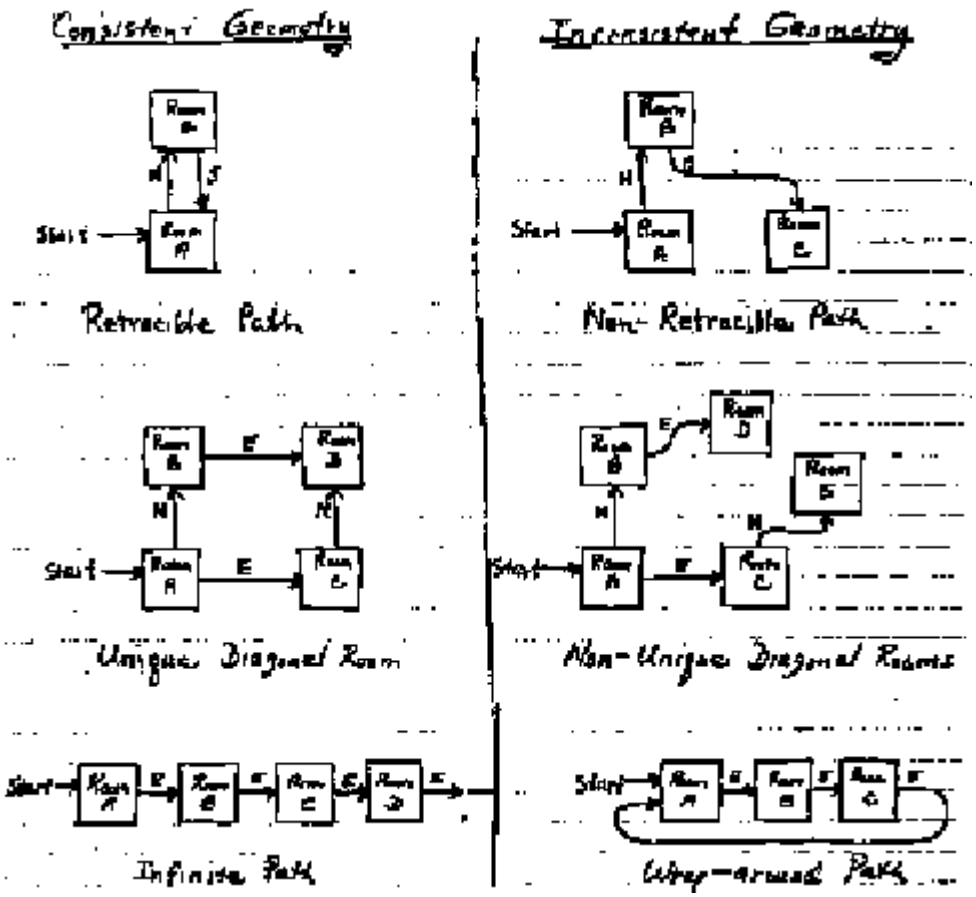

Figure 3-6
Consistent versus Inconsistent Geometry

In moving through space, common sense suggests that the space passed through is still there, and that, if necessary, one should be able to turn around and retrace one's path back through that space. Of course, this is not always true in real life: there are one-way doors that swing shut and lock, and trap-doors that one can fall through and not be able to climb out of. If the player can go one room north, then go south from that room and be back where he started, he has retraced his path. However, if the room-to-room south link doesn't take him back where he started, there is no backing up -- he is on a one-way, non-retracible path. This non-retracability is a device the designer may use to construct a trickier, more confusing maze. Confusing the player to a certain degree is a quite proper objective for the game designer, because it is essential that a game be challenging. However, too much illogical trickiness can frustrate and irritate the player. In the video game Adventure, I chose to let the player be always able to retrace his path.

Does going one room north, then one room east get to the same place as east-then-north? The answer is either yes or no, depending on how the rooms are linked.

On a flat plane, going east forever will never bring one back to his starting point. However, the mathematical idea of an infinite plane implies infinite area, and it is hard to find truly infinite things in the real world. The surface of the earth, which is pretty flat, wraps around to meet itself eventually. The limited memory of a computer would have difficulty representing an infinite number of rooms. In a finite network of rooms, any path must eventually return (wrap around) to some previously encountered room, assuming the path is never blocked by walls.

The pattern of interconnection within a network of rooms is called topology. Topology has to do with what is connected to what. The four mazes of Adventure are all unrealizable in the flat. They wrap around in various strange ways (See Figure 3-7), and thus have interesting topologies.

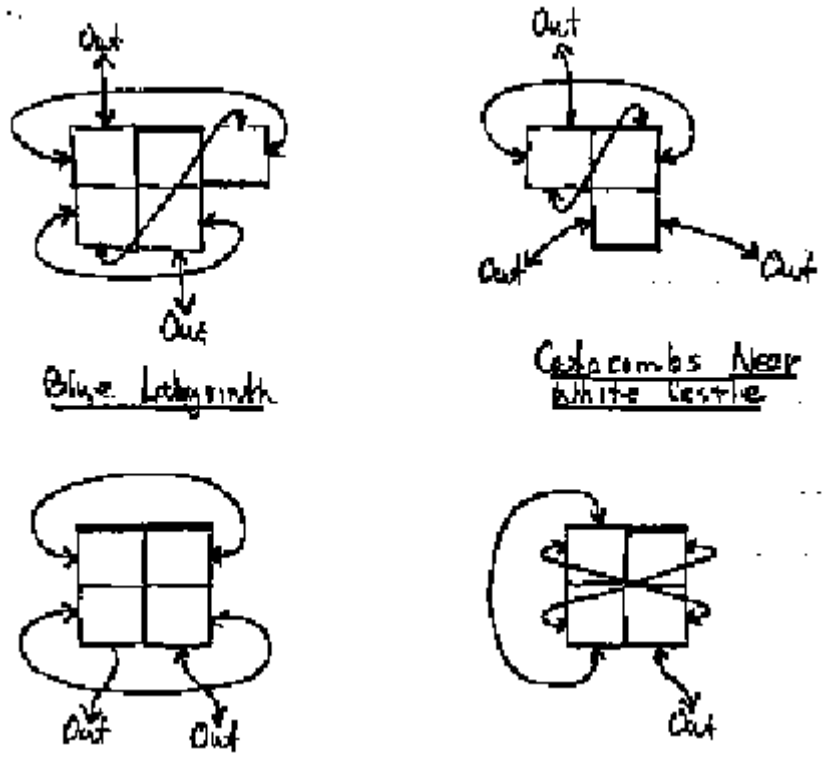

Figure 3-7
Comparison of the Room Topologies
of the Four Mazes in Atari 2600 Adventure

The Blue Labyrinth

The Blue Labyrinth in Adventure has two features which make it different from mazes printed on paper: inconsistent geometry and partial view. Figure 3-8 shows how the five rooms are interconnected to form the Blue Labyrinth. The maze is more confusing than the diagrams suggest because only a part of the maze (one room) can be seen at any one time. Viewing the diagram, the eye can rapidly follow maze paths and identify deadends; but in playing the game, the cursor must move along each path to explore it, and laboriously retrace from deadends. Exploring is not only slower, but the explorer must rely on his memory of the maze-rooms he has passed through in order to form a mental model of the whole maze. Players find the Blue Labyrinth quite confusing at first -- more so than would be expected from the number of its forks and different passages -- and the principal source of this confusion is that the player gets only a partial view of the maze, never seeing the whole. One is reminded of the story of four blind men examining an elephant, forming different conclusions about what kind of beast it was from feeling its trunk, tusk, foot and tail. It is hard to reconcile several partial views of something into a coherent global picture.

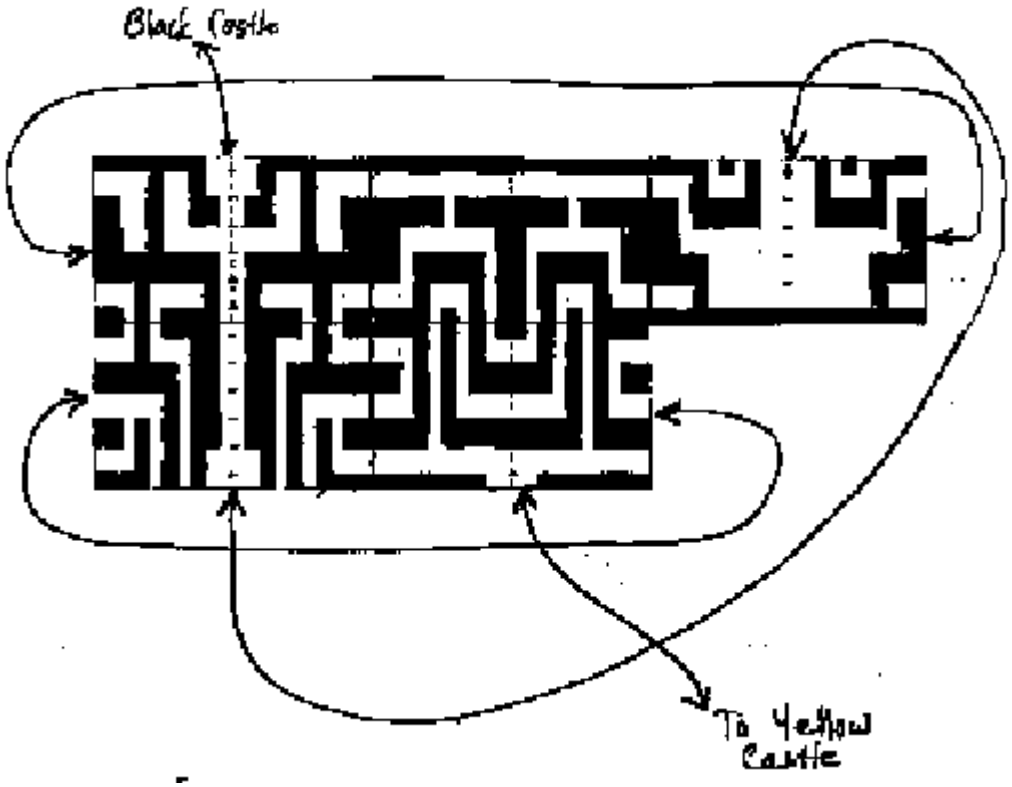

Figure 3-8
Room Topology of the Blue Labyrinth in Atari 2600 Adventure

To add to the confusion, unless normal assumptions about spaces are abandoned, no coherent model of the Blue Labyrinth exists. This is because the Blue Labyrinth has several wrap-around paths and non-unique diagonal rooms. A player remarked that he could learn paths through the maze from place to place, but could never get a picture of the whole thing in his mind.

These inconsistent maze geometries are confusing because the player's experience with mazes (usually printed on flat pages) leads him to expect more flat mazes. He attempts to make a mental model which incorporates the maze rooms he has seen into a flat map. As he explores, contradictions occur in his mental flat map, but the assumption that maps are flat surfaces is very deep and hard to challenge. Experience exploring real-world mazes at least offers a clue -- the slope changes -- when surfaces are not flat. But the Blue Labyrinth offers no such clue.

The Red Maze

Inside the White Castle lies the Red Maze. (See Figure 3-9). The distinguishing feature of this maze is that it is composed of two disjoint sets of passages that are intertwined with each other, but do not connect. The player enters Section l of the maze through the door to the castle, and to be able to get into Section 2, he must bring the bridge object into the castle and use it to cross one of the maze-walls which separate the two sections.

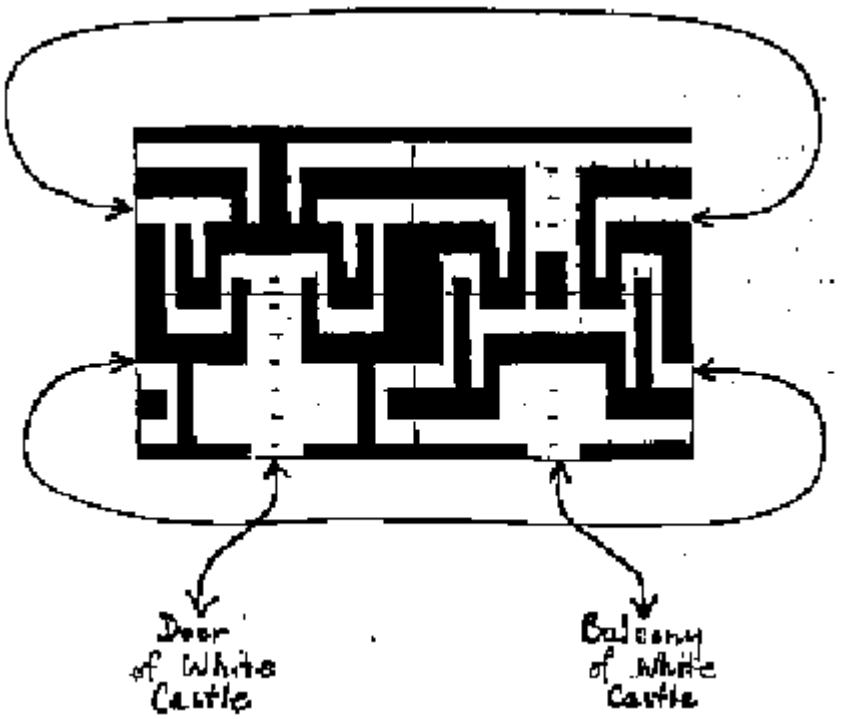

Figure 3-9
Room Topology of the Red Maze
Inside the White Castle

Section l extends through only three of the four rooms of the Red Maze; without using the bridge, the player cannot get a glimpse of the fourth room and whatever useful objects it might contain. In Game 2 of Adventure, the key to the Black Castle starts out in this hard-to-get-to place.

The topology of the Red Maze is simpler than that of the Blue Labyrinth. The Red Maze wraps around in the horizontal dimension, and going down from either of the two lower rooms exits from the castle. One of these exits is the normal one through the door of the castle, but the other exit, from the hard-to-get-to room in Section 2, leads to the "balcony" of the White Castle. The link from the Red Maze to the balcony is one-way, however; it is not possible to go back into the Red Maze from the balcony. This violates my principle of making all paths retracible, and was a mistake -- a rejected earlier idea that was never expunged. (Once a product is released, harmless bugs like this one are often redefined by the marketing people as "features." The manual for Adventure explains away its bugs under the heading "Bad Magic.")

Catacombs

In the original text Adventure, once the player got a couple of rooms into the cave, he got this message:

```
It is now pitch dark.  If you proceed you will likely
```

```
fall into a pit.
```

The solution was to light the "shiny brass lamp" picked up earlier. Atari 2600 Adventure has a maze which works in analogous fashion: orange lamp-glow penetrates a short distance into the surrounding gloom to expose nearby maze walls. Mazes of this type are called catacombs. Figure 3-l0a diagrams the walls of a typical maze-room in the catacombs, and the dotted line around the cursor shows how far the lamp-glow reaches. This illuminated area around the cursor is equivalent to the circle of radiance thrown out by a lantern, but in this case, a smooth circle being impossible, the circle of radiance is square. Beyond the illuminated area near the cursor, walls cannot be distinguished from passages. Figure 3-l0b shows the screen image resulting from the maze position diagrammed above it. The static image does a poor job of conveying the feel of being in the catacombs; the cursor can move about the screen, bumping into unsuspected walls, and the orange firelight surrounds the cursor wherever it goes. It is like shining a flashlight around in a cave.

(a) diagram of invisible part of maze

(b) actual screen image

Figure 3-10
One Maze-Room in the Catacombs

Just as seeing a single room of the Blue Labyrinth is a partial view of its 5-room entirety, the image of the maze-walls near the cursor is a partial view of a single catacomb room. The lamp-glow covers about a tenth of the screen . The two catacomb mazes in the game consist of three rooms and four rooms. So, in the catacombs, a partially viewed room, if the player could imagine it entire, is itself only a partial view of a multi-room maze. The smaller the individual views are relative to the whole, the greater the difficulty in assembling them into an overall picture. The Blue Labyrinth, with its five rooms viewed one at a time, has a fractional view ratio of l:5. By comparison, the 3-room catacombs near the White Castle, with each room seen one tenth at a time, compounds fractional views of l:3 and l:l0 to yield an overall ratio of l:30. Thirty different images must be remembered (or sketched) and joined together correctly to get a picture of the entire maze.

The interconnections among the three rooms of this maze are shown in Figure 3-ll.

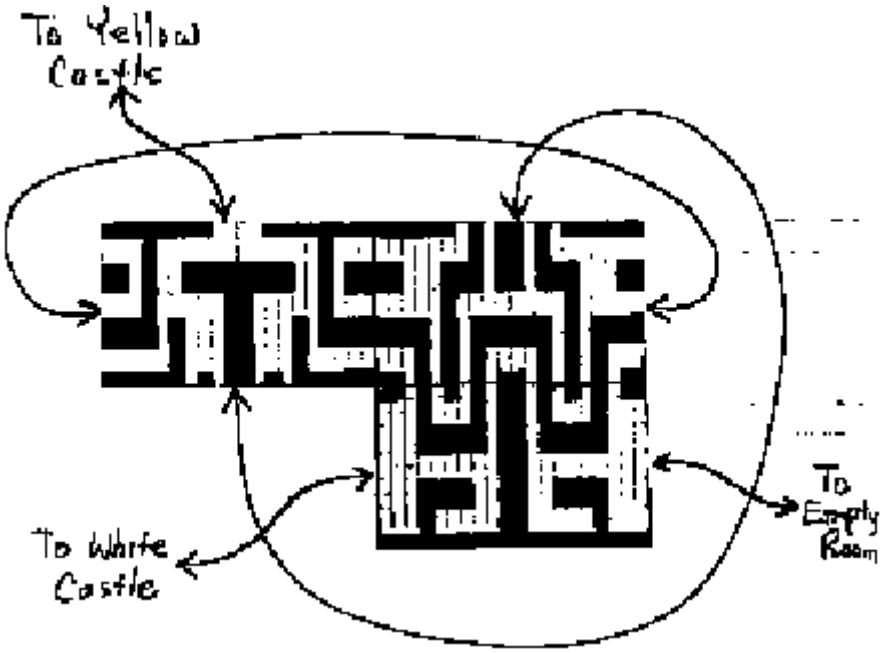

Figure 3-11
Room Topology of the Catacombs Near the White Castle

The Catacombs Inside the Black Castle

The maze inside the Black Castle is a tricky one. It is a catacomb-type maze, with the lamp-glow surrounding the cursor allowing only a partial view of each room. Four rooms compose the maze, yielding a view ratio of l:40. In addition, the topology of the maze is complicated. (See Figure 3-l2.) Like the Red Maze, this maze has two disjoint parts: the bridge is needed to get from one section to the other. But unlike the Red Maze, where the two sections were of similar size, in this maze the isolated section is a tiny little chamber at the bottom of one room. The existence of this chamber is not obvious because when the player's cursor is beside the chamber, but not in it, the orange lamp-glow does not extend far enough for the player to see that the chamber is surrounded by maze-walls on four sides. The chamber (partially viewed) seems to be just the deadend of one of the other passages. And yes, there is an interesting object -- the Gray Dot -- hidden within this chamber.

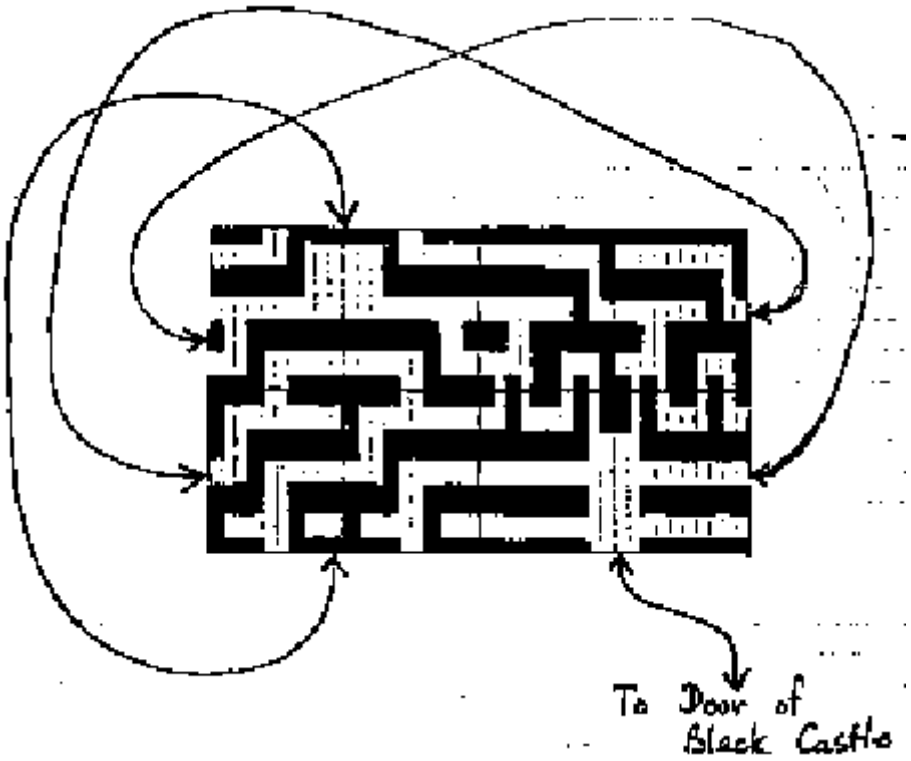

Figure 3-12
Room Topology of the Catacombs
Inside the Black Castle

The dot was not needed in the normal play of the game. Being gray, it was invisible until it was picked up. It was not mentioned in the game's manual because Steve Harding, who wrote the manual, didn't know it existed. The dot's purpose which had to be discovered by trial and error, was to allow passage through a certain side-wall into a secret room. In this deviously hidden secret room, I affixed my signature, in flashing lights: "Created by Warren Robinett."

I did this in the tradition of artists, down through the centuries, identifying themselves as the authors of their own works. Atari imposed an irksome anonymity upon its designers, so subterfuge was required to put one's mark upon a game. I kept the secret of the signature room to myself for a year (not an easy thing to do) in order to avoid being forced to expunge it from the program, and also to test experimentally whether anyone could, on his own, discover such an obscure thing. I really wasn't sure that it would be discovered, but thought that if several hundred thousand cartridges were manufactured, then someone, somewhere, would find the dot and the secret room. I remembered, from when I was in high school, the rumors that Paul McCartney was dead, and how people played Beatle records backwards, searching for secret messages.

The Adventure cartridge was manufactured and marketed, and the secret room was discovered. A l5-year-old from Salt Lake City wrote to Atari, explaining with a detailed diagram how to get into the secret room. By then, I no longer worked for Atari, so they had a couple of their designers track down the part of the game program that produced the secret room. Brad Stewart, who located the offending code, said that if he was assigned to change the program, he would replace "Created by Warren Robinett" with "Fixed by Brad Stewart." Ultimately, Atari blessed the whole idea, referring to hidden surprises in their games as "Easter eggs."

The major innovation of Atari 2600 Adventure is the idea of moving a cursor through a network of screens connected edge-to-edge. This idea made it possible to make a video game which was at the same time an adventure game, by identifying the network of screens with the adventure game's network of rooms. The action of the game could therefore take place in a much larger and more interesting space than the single screen of most of the then-current video games. It was natural to adopt the small movable shapes provided for in the video game hardware to be adventure game objects. Woods had established in his text adventure that objects were tools for getting past obstacles. My idea for a sequel to Adventure was to allow tools to be combined in order to solve more complex problems. I thought of this as building machines. The idea evolved as I worked on it. The end result, three years later, was a game in which tool-objects (sensors, logic gates and others) could be combined to make machines which made things happen in the game in response to conditions detected by the sensors. This game was called Rocky's Boots.

Chapter 4

Adventure as an Educational Simulation: *Rocky's Boots*

Building Machines in an Adventure Game

Convinced that the animated adventure game was virgin territory, begging to be explored, I made notes for a new adventure game, in which the player would build machines to defeat the monsters. The idea was that the player possessed insufficient powers to achieve much success in the game on his own, but by building machines, he could overcome the monsters and surmount obstacles to win the game. Using objects as tools in an adventure game was an established idea, and combining and connecting tool-objects into "machines," which might, in turn, be used as components for even larger machines, extended this idea. My model was the Tinkertoy set, in which little pieces are plugged together to form larger structures. I decided that logic circuit elements -- AND gates, OR gates, NOT gates and flipflops -- could process the information provided by "sensors" that detected conditions in the game, and the resulting logic signals could control "effectors" that did things within the game world. Thus, for example, a monster sensor could be plugged into a logic-controlled cannon to make a monster booby-trap. From a small number of sensors, effectors and logic elements, quite a wide variety of machines could be built. In progressing through the adventure game, surviving by using the materials at hand to build tools to solve the problems he faced, the player would be reenacting the history of man, who began by clouting bears on the head with sticks and rocks, and later built great cities out of wood and stone.

In the spring of l980, I submitted a proposal to Personal Software (later renamed Visicorp) to develop an "Animated Adventure Game Program" for the Apple and Atari computers. In spite of my promise of "large lumbering dragons, slow slithering snakes, faithful dogs, and cheerful singing birds," they instead chose to fund a different adventure game (Zork). However, they did introduce me to Ann Piestrup, whose proposal to develop educational software they had also turned down.

A few months later, Ann, an educational psychologist, got a grant from the National Science Foundation to do research and software development regarding "Early Learning of Geometry and Logic Using Microcomputers." The target audience for this microcomputer software was gifted second- and third-

graders. My ideas about logic machines in adventure games fit within the scope of the grant, and the prospect of being able to develop these ideas for a year enticed me into joining the project. Mathematics educator Teri Perl rounded out our team. Teri, Ann and I explored the genre of the graphical adventure game, with its animated objects and network of rooms, as a medium for educational software. I implemented graphical adventure game software on the Apple II computer, adopting the conventions of joystick movement, rooms, walls, picking up objects, and so on, that I had used in the Atari 2600 Adventure cartridge.

I used adventure game objects to make some simulated logic elements. For the shapes of these objects, I chose the standard symbols used by electrical engineers in their diagrams of computer circuits. These symbols are called AND gates, OR gates, and NOT gates. (See Figure 4-l.) However, since these symbols appeared on an animated color display, rather than paper, it was possible to show the state of each circuit element. Logic circuits allow only two states, which are called "1" and "0" or sometimes "true" and "false." Orange and white worked well to show the logic state of each gate in a circuit diagram, with the whole circuit drawn on a black background. The logic elements could be interconnected to form a logic circuit, and as it performed a logical calculation, its components flickered between orange and white as they showed the logic signals that propagated through the circuitry.

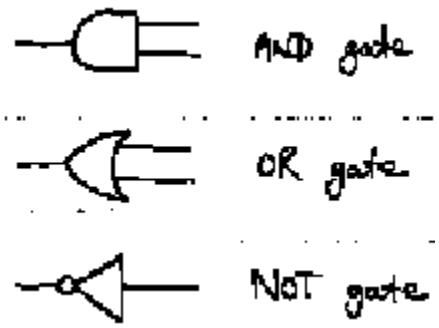

Figure 4-1
Logic Element Symbols

The grant funding lasted for a year. At the end of that time, we had a very interesting logic circuit simulation. The circuits were embedded in an adventure game, but it was not a good game. The machines a player could build were rather ineffectual against the game's voracious alligators. Kids just ignored the circuits and manually dodged alligators while they hunted through the mazes for treasure. Obviously, more work was needed. Venture capital financing enabled us to keep going. Ann, Teri and I started The Learning Company, together with Leslie Grimm, who had written several learning games already, and Jack Smyth, who became president.

Two things were added to the logic circuit simulation to make it into a useful, saleable product: a tutorial which explained how to use the circuit simulation, and a game integrated with the simulation so that the player could build circuits in order to win the game. This was called Rocky's Boots, after the boots in the game, which were activated by logic signals, and Rocky the Raccoon, who danced a jig when the player solved a logic puzzle by building a correct circuit. With Leslie Grimm's help during its final few months of development, I finished Rocky's Boots in September, l982.

Rocky's Boots was not the adventure game I had originally wanted to create. It was not an adventure game at all. Rather, it was educational software set in an adventure game world of rooms and objects. It used the connected rooms like book pages to make an explanation, a tutorial, about the circuits. It used adventure game objects as movable components, to be plugged together to make the circuits themselves. And finally,

I contrived a rather bizarre game of targets, sensors, and a target-kicking boot in order to create a game situation in which the player could employ his new skill in building logic circuits to solve a wide variety of problems.

Circuit Components

Each component in a real digital logic circuit is always in one of two states; physically, two different voltage levels occur in the wires of the circuit. These two states are referred to as true and false, or logic l and logic 0. In the graphical depiction of the simulated circuit, orange and white were used to show the logic states of each circuit element. As the circuit progressed through a computation, the user could observe signals as they propagated through the circuit. The explanation given was this:

```
These machines are like the lights in your house.
Electricity turns them on.  But in this game you can
see the electricity.  It is orange.
```

In the tutorial, I chose to talk about electricity, which kids (and adults) know of as invisible stuff that flows inside of wires, rather than signals, a concept that struck me as difficult to explain to a l0-year-old. This made possible a wonderful metaphor: the electricity -- the orange stuff -- was liquid fire that flowed through the transparent pipes which composed the circuit. I was pleased, one day, to hear a young user refer to a wire in the simulation as a "pipe."

Signals flow in and out of real digital logic components along wires, and some of these wires are inputs (accepting signals) and others are outputs (producing signals). In the simulation, I invented symbols to identify inputs and outputs (See Figure 4-2). Two components could be connected by attaching the output of one to the input of the other. The electrical signal flows one-way across the connection: from the output to the input.

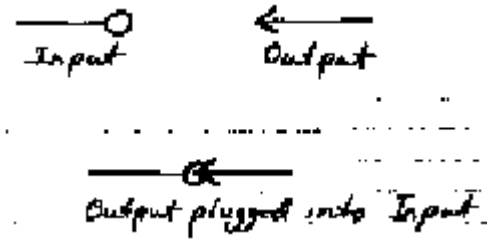

Figure 4-2
Symbols for Inputs and Outputs

The logic circuit elements in the simulation are adventure game objects, and they exist within the network of adventure game rooms. These circuit-objects could be picked up, moved around, and dropped, as is standard in adventure games. When two circuit-pieces were plugged together, the resulting circuit could be picked up and moved as a single object. There was a means of unplugging (the knife object) which restored the two circuit-pieces to being separate objects.

There were more than a dozen kinds of circuit components in Rocky's Boots, and these fell into three categories: sensors, effectors, and logic elements. Sensors detected conditions in the game, and therefore produced only output signals. Effectors, on the other hand, could cause actions to occur in the game, but did this under the control of an input signal. Logic elements had both inputs and outputs. A logic element performed some sort of transformation on its inputs, passing the result on as its output signal. The obvious arrangement of sensors, effectors, and logic into a circuit was for the sensor to feed signals into a logic network, which in turn produced signals to control some effectors. This is much like a living creature, where the eyes and other senses send information along nerves to the brain, which decides what to do based on what it perceives, and then sends signals to the muscles in order to take action.

The logic elements available for making these simulated brains were AND gates, OR gates, NOT gates, flipflops, delays, clocks and wires (See Figure 4-3). Flipflops and delays were memory elements. Clocks provided timing signals. Wires were for conveying signals from point to point, unchanged. The real logic operations were performed by the AND, OR and NOT gates. A NOT gate produced an output which was the opposite of its input. An AND gate turned its output on (orange) only if both of its inputs were on, whereas an OR gate turned its output on if either of its inputs were on.

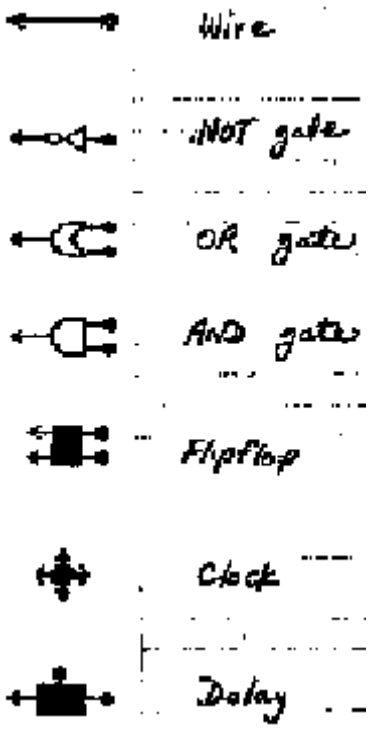

Figure 4-3
Logic and Memory Elements

The wires in Rocky's Boots were criticized occasionally for being one-way when in real circuits wires can carry signals in either direction. This is a valid criticism, because there are some computer circuits (bi-directional data busses) that use physical wires in two-way fashion. However, to deal properly with two-way wires would have required simulating the circuit at the impedance level rather than, as was done, at the (simpler) logic level. Another way that the circuits of Rocky's Boots deviated from the reality of physical circuits was that if the outputs of two real chips were hooked together, one of the chips might be "fried" (a

technical term for overheating to the point of no longer working). Rocky's Boots didn't allow two outputs to be plugged together, and thus missed a chance to faithfully mimic that aspect of real circuits.

Sensors varied according to what they sensed. There were green-sensors, blue-sensors and purple-sensors, and also sensors which detected targets shaped as squares, circles, diamonds and crosses. Each sensor sensed the presence or absence of one color or shape, and produced a corresponding on-or-off logic signal at its output. There was also a different kind of sensor in the hidden alligator room, an alligator radar, which sensed the relative position of the alligator.

Effectors caused things to happen in the game. When the boot was activated (by turning on its input), it flew across the screen to kick a moving target. When the thruster was turned on, it moved around the circuit it was part of. The bopper was useful for knocking pesky alligators out of the way. The clacker and On-Off sign gave auditory and visual feedback to the player. (Figure 4-4 shows some sensors and effectors.)

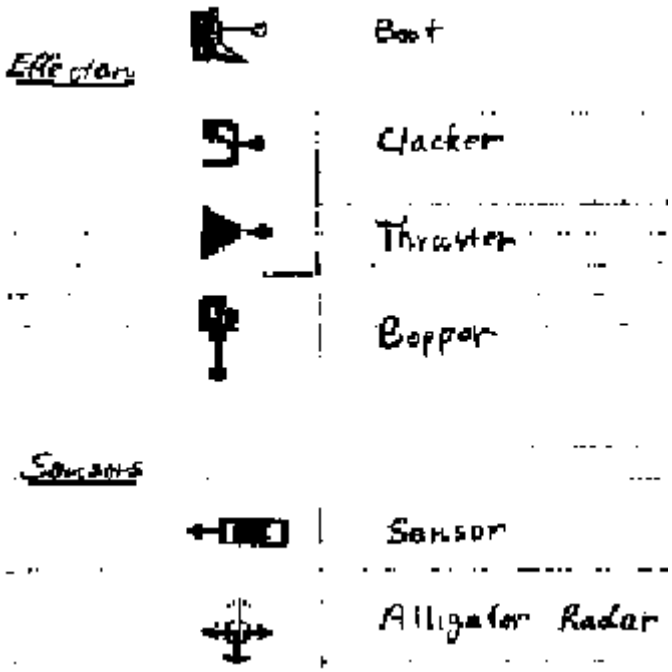

Figure 4-4
Effectors and Sensors

Kicking Targets

The games in Rocky's Boots all had the same format. The player's goal was to build a logic circuit which selected a specified subset from a group of "targets." The targets had attributes of color (blue, green and purple) and shape (square, circle, diamond and cross). The targets would pass by an array of sensors which scanned each target for the presence of individual attributes. A green-sensor, for example, would turn on when it scanned a green target. A circle-sensor responded to targets which were circles. The player needed to use the signals produced by the sensors as inputs to the circuit he built. For example, if a green-sensor

and a circle-sensor were plugged in to the two inputs of an AND gate, the AND gate turned on only in response to a green circle being scanned. If, in turn, the output of the AND gate was plugged in to the logic-activated boot, then when the AND gate turned on, the boot would fly across the screen to kick the current target -- the green circle.

The sensors detected conditions within the game -- the presence of targets with certain attributes; the boot was an effector that produced an action in the game world -- kicking a target was the means of selecting it. The targets which were meant to be kicked were assigned positive point values, and the other targets received negative point values. Thus, to achieve the maximum score, the player had to build a circuit that kicked exactly the intended subset of the targets. Figure 4-5 shows the sensors, circuit and boot, with a target being scanned as it passes near the sensors.

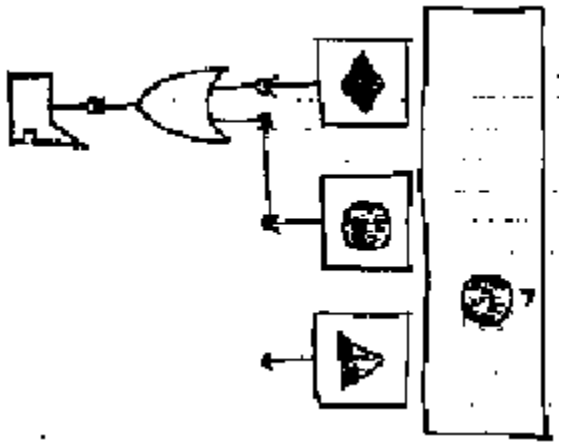

Figure 4-5
Boot, logic circuit, sensors and target

This game was hard to explain because there was no consistent fantasy or metaphor behind it. With its electricity, targets, and boots, it certainly could be called a mixed metaphor. However, it did serve the purpose of providing a format in which a wide variety of logic circuit design problems could be posed. It worked fine for problems in combinatorial logic, which encompasses the AND, OR and NOT logic elements. Sequential logic deals with a broader class of elements, adding time-dependent elements such as flip-flops and delays. The target-kicking format was well-suited to posing sequential logic problems, too, because the targets were processed sequentially, moving past the scanning sensors one at a time.

An Interactive Tutorial

There was a definite need for the player to either discover, or have it explained to him, how the logic elements worked, and related matters, such as how to connect them into circuits. An adventure game in which the player discovered, on his own, how logic gates worked, and then built circuits with them to solve problems would, of course, have been wonderful. However, using explicit verbal explanations was a direct route to getting someone to understand the simulation.

In this implementation of the graphical adventure game idea on the Apple computer, it was possible to put alphanumeric text on the screen. The text could be considered as part of the room, with each room containing its own set of paragraphs or sentences. The cursor could be moved from room to room, with the user viewing each new screen full of text., in the same way that a reader moves from page to page in book. The method used to move from "page" to "page" on the screen -- namely, the normal joystick-controlled method of moving from room to room -- had the advantage, like a book, of being self-paced and of allowing the reader to back up and look at earlier material.

The adventure game rooms filled with text emulated book pages; the adventure game objects were used to simulate digital logic circuits. Since the objects existed in the space defined by the network of rooms, working logic circuits could be freely intermixed with explanatory text. Thus, for example, a phrase of text could introduce the AND gate logic element, with a simulated AND gate appearing immediately below the text, available to be experimented with by the player. (See Figure 4-6.) A sequence of rooms could introduce a series of ideas, with working parts inserted every so often, upon which to try those ideas out. The adventure game rooms had walls that blocked the movement of the cursor, and these walls were useful in channeling the movement of the player's cursor along a linear path.

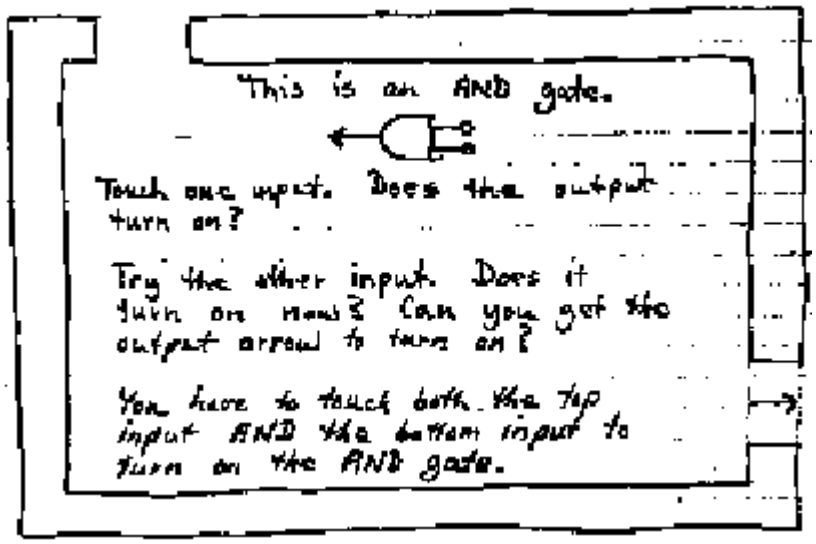

Figure 4-6
Text and working circuit elements are interleaved in an interactive tutorial

The linear sequence of rooms, with no alternate paths, seemed to work best for material of a tutorial nature, where ideas needed to be introduced in order. I tried a fork in the path in one tutorial in Rocky's Boots: players with joysticks connected to their computers were asked to go one way, and players without joysticks another way. This seemed to cause more problems than it solved. Wondering what they would be missing, players sneaked into the wrong path, and then became confused when they reached the point where the two alternate paths rejoined. In a linear, non-branching series of rooms, the author can control the order in which ideas are presented.

Computer-assisted instruction had been criticized for doing nothing more than mimicking books; its programs were derided as computerized "page-turners." Critics quite rightly insisted the computer should be used to do something that was not possible with printed material. The combination of screens of text with animated, interactive illustrations seemed a step in the right direction.

A network of adventure game rooms was effective for simulating pages from a book because the adventure game has a simple, consistent model of space. Book pages exist in different locations in space in a real book, and are hinged together on one side. We learn, at a young age, how to thumb from page to page in a book. This is moving through book-space. This action does not interrupt the act of reading, or force the reader to look momentarily elsewhere. Book pages simulated by adventure game rooms have similar advantages: the means of moving through space, and hence from page to page, is part of the adventure game. When a joystick is used to control the cursor's motion from page to page, this motion can be controlled without the need to look away from the screen. For children, and other people who are not good typists, it is distracting for them to have to periodically interrupt their reading in order to hunt on the keyboard for the key that moves them to the next page.

Interactive Graphical Simulation

Electricity is hard to understand because it is invisible, and, in addition, because it acts very quickly: a single electronic pulse travels through a circuit and does its work in a few billionths of a second. In simulation, happily, time can be slowed down to a convenient rate. Choosing a simulation time rate is a compromise: if the simulation moves too slowly it is boring and tedious, but if events succeed one another too rapidly, then the viewer becomes confused about what caused what. One event causing another one occurs, in normal experience, with the two events near each other in space, and with the evoked action immediately following the provoking action in time. The graphical format of the logic simulation con conveniently show spatial relations such as nearness, and, to achieve maximum clarity, the designer's goal must be to make the delay between an event and its cause long enough to discern which came first, and short enough to make it evident that the two events are related. In such a simulated machine, if the different states and positions of all its parts are clearly discernable, and if the viewer can see how changes in one section of the machine cause other parts to change, then simply watching the machine in action is very informative. A transparent model of a car engine, with the pistons moving up and down, is informative in a similar way. Watching a well-timed graphical simulation can strongly suggest relations of cause and effect among the elements of the simulation.

If, furthermore, the user can experiment with the parts of the machine, to test out his theories of how the parts work, and what causes what, then the user has a very powerful tool for understanding the phenomena being simulated. He can experiment and discover how the parts work.

Ann Piestrup advocated the idea that "the computer never tells the child that he is wrong." The interactive tutorials provided a risk-free environment for experimentation. It was Leslie Grimm's idea to provide "practice rooms" at the end of each tutorial, to encourage experimentation. Still, a learner needs feedback at some point to test himself, and know if he is getting the idea: the win-lose outcomes of the games provided this feedback. Even in the games, failure to win was mild and reversible: an incorrect circuit just did something other than what the player intended, and it could be rebuilt easily, with the player proceeding toward a correct solution by trial and error.

If the user can combine the simulated elements into more complex structures, this helps him test his understanding. Children build from a set of parts with Tinkertoy sets, Lego bricks, Erector sets and simple wooden blocks. Using the computer, building is done with little shapes on the screen rather than little physical objects. Rocky's Boots is an electronic Tinkertoy set.

The use of orange and white in the simulation to show the on-or-off state of each component makes the behavior of a circuit visible. The ideas of psychologist Jean Piaget suggest that a child is able to grasp concepts with concrete representations first, and abstract ideas later in his life. Orange and white are concrete. By contrast, digital logic circuit design has traditionally been taught in a much more abstract form using a branch of mathematics called Boolean logic. Boolean logic uses an algebraic language of logic equations, and symbols standing for logic signals. It is not surprising that grade-school children didn't

or couldn't use Boolean logic. It is interesting, however, that these children could understand the underlying logic circuits when the circuits were given a concrete representation.

The graphical images I chose to represent the AND, OR and NOT gates were the same symbols that electrical engineers use when they design computer circuits. Thus, the logic circuits that the player designs on the screen look just like the circuit diagrams in the back of computer reference manuals. But, there is a difference: the on-screen circuits work. These circuits are both symbolic diagrams and working, fully functional logic circuits. The abstract with its symbols and the concrete with its immediately perceivable objects approach rather closely here. I call these concrete abstractions symbol-gizmos.

The graphical simulation of logic circuits in Rocky's Boots was important both because of its topic, and because of its general method of presentation. The topic, digital logic circuits, is basic to any understanding of computer hardware. Every computer in existence is made from the simple logic and memory components presented in the simulation. An understanding of AND gates, OR gates, NOT gates, and flipflops is a firm foundation from which to reason about the workings of a complex piece of computer hardware, just as in chemistry, a knowledge of the properties of atoms is a prerequisite to understanding the behavior of molecules.

The method of presentation of the logic circuits is as an interactive graphical simulation. Simulation offers a phenomenon from the world faithfully mimicked (but safely contained) in the computer. Graphical simulation brings the eye's great powers of analysis to bear in aiding the understanding, and in grasping the relations of cause and effect which govern the phenomenon under study. Interactivity lets the user test his tentative understanding by tweaking parts of the simulation and seeing if it behaves as expected.

Rocky's Boots simulates several different things. It uses the adventure game metaphor of connected rooms to simulate a large space. By dropping pages of text along a path through the rooms, it is easy to simulate a book. Adventure game objects are used to simulate individual logic elements, and by providing a means plugging these elements together, it is possible to simulate logic circuits.

Interactive graphical simulation has been explored, in a preliminary way, in a few areas. Much uncharted and fertile ground beckons. Those microcomputer software products which have been most interesting, useful, and successful -- electronic spread-sheets, word processing, and video games -- may be fairly described as interactive graphical simulations. A spread-sheet program like Visicalc simulates a financial system. A word processing program simulates scissors, paste, paper, and a typewriter. Video games simulate spaceships, missiles, bouncing balls, and other things.

Many provocatively complex phenomena await interpretation into the interactive graphical format -- trains and other vehicles which move cargo through spaces, kayaks in swirling river currents, planets orbiting their stars, competing creatures in evolving ecologies, visible melodies smeared upon harmonic wallpaper, looping programs in throbbing execution, and human thought darting across a tangled network of knowledge.

Chapter 5

Getting Ideas

What makes an idea for a computer game a good one? People can like a game for various reasons -- because they like competition, learning a new skill, problem-solving, exploration, or controlling something's motion, or because they are engaged by the fantasy or beauty of a game's environment. To evaluate an idea for a game, one must estimate the likelihood of turning the idea into an appealing game. This task has two parts: forecasting what will interest people, and judging whether an idea could be implemented as a working program on a particular computer.

Human interest can be sparked by a vast range of topics, but the interactive, visual form of the computer game medium is more suitable for some topics than others. For example, people are interested in other people's looks, and thus arose the art of portraiture in painting and photography. However, it is hard to imagine a satisfactory portrait of a particular person as an interactive computer game. (Hmmm. An interactive portrait -- perhaps the player's input could control the facial expression.) On the other hand, certain aspects of war can be portrayed quite effectively in a computer game. The general's strategic decisions, the map of a developing battle, the pilot's out-the-window view, and the explosion of the falling bombs each have been convincingly depicted in a game. It would seem, barring advances in interactive portraiture, that the computer game medium shows war better than faces. At any rate, some topics have been demonstrated to make interesting computer games, and other topics are still waiting to be explored. One way to prospect for good game ideas is to think about the nature of the interactive computer game.

A computer game is a simulation. The game program models the interactions of distinct entities defined by numerical attributes. For example, many video games simulate physical objects moving and colliding. Pong simulated a bouncing ball. Asteroids simulated a thrusting, turning spaceship. Pole Position simulated a race car and track. Other video games have simulated sports, like football and tennis, and card games, like Blackjack. Adventure games simulate objects in a network of rooms.

The real world offers a vast set of phenomena to stimulate -- animals behaving, plants growing, structures buckling, traffic jamming, snowflakes forming. Any process is a candidate. Every verb in the dictionary suggests an idea. Making a simulation is a process of abstracting -- of selecting which entities and which properties from a complex real phenomena to use in the simulation program. For example, to simulate a bouncing ball, the ball's position is important but its melting point probably isn't. Any model has limitations, and is not a complete representation of reality.

Idea Book

Ideas pop up while you're doing something else. The best way to nurture and ultimately harvest these spontaneous blooms is to write them down. An idea book, built up from scribbled down ideas, is a wonderful resource. It is a granary, a storehouse from which to draw in times of drought.

A good computer game is often based on several ideas that work well together. One idea might specify a metaphor or theme for a game, another might be about a method of displaying objects in the game, and a third idea might be about how input from the player would affect the game's action.
These ideas would not necessarily come to the designer all at the same time. Ideas seem to require a gestation period -- a time of mulling and musing -- before they become workable. Written notes are a good way to make sure no insights are lost along the way.

For a game idea to be feasible, the designer must have a plan for implementing it on a particular computer. A general plan is good enough in the beginning. Such a plan should specify how the entities in the game-simulation will be represented with data, and what routines must be written to perform the simulation. There should be reason to think that this program and its data can fit into the memory space of the target computer, and that the program will be fast enough not to offend human patience. Having settled on an idea, the task of the programmer is to take the idea and turn it into a working program. My starting point is always an idea written down in a few lines on a piece of paper. For example, Figure 5-1 shows my general plan for implementing a graphical adventure game of rooms and movable objects on the Apple II computer. (This software was the foundation on which *Rocky's Boots* was built.) The plan includes, first, a scheme for organizing data in memory (in this case, for describing rooms and objects), and second, descriptions of routines that will operate on this data.

As a second example, Figure 5-2 shows my notes for the idea of simulating logic gates with adventure game objects. These notes give some idea of how ideas come -- as related thoughts but in no particular order. In contrast, the notes of Figure 5-1 are really an implementation plan, which is the next stage beyond a raw idea. An idea ("Wouldn't it be neat if you could...") often says nothing about how the program to do it would be structured. It is up to the programmer to come up with a plan for implementing his idea.

Not every idea can be implemented. The resources of memory and processing power are limited on every computer, and particularly so on personal computers. Experienced programmers keep the speed and space limits of their computers in mind as they plan. The notes of Figure 5-1 include some timing estimates (in milliseconds) for the two slowest routines.

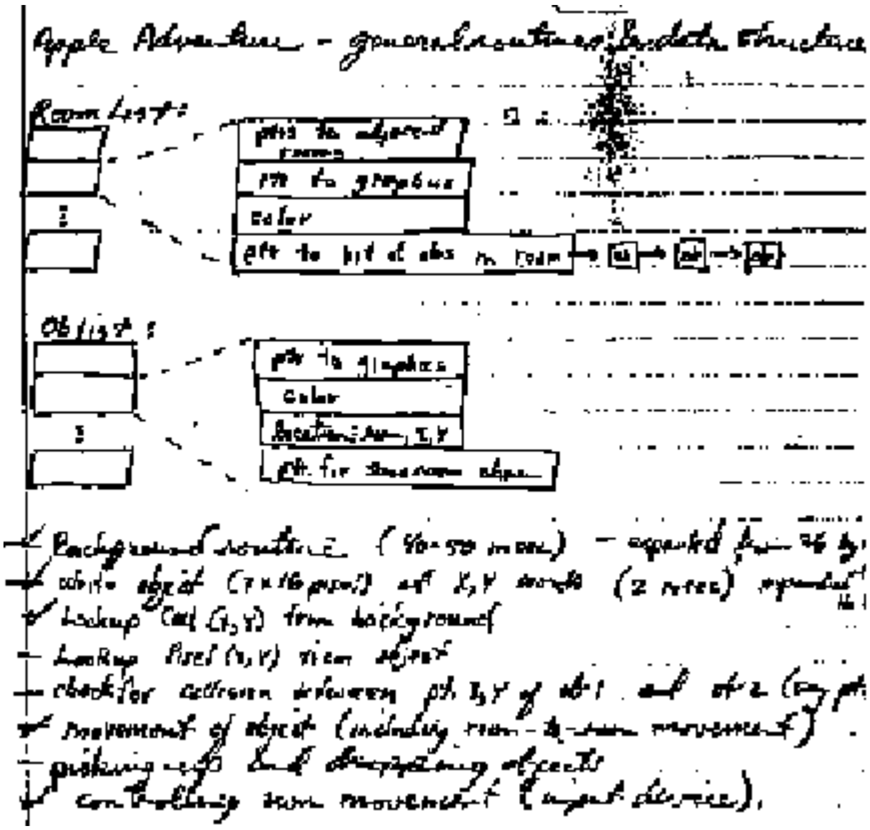

Figure 5-1

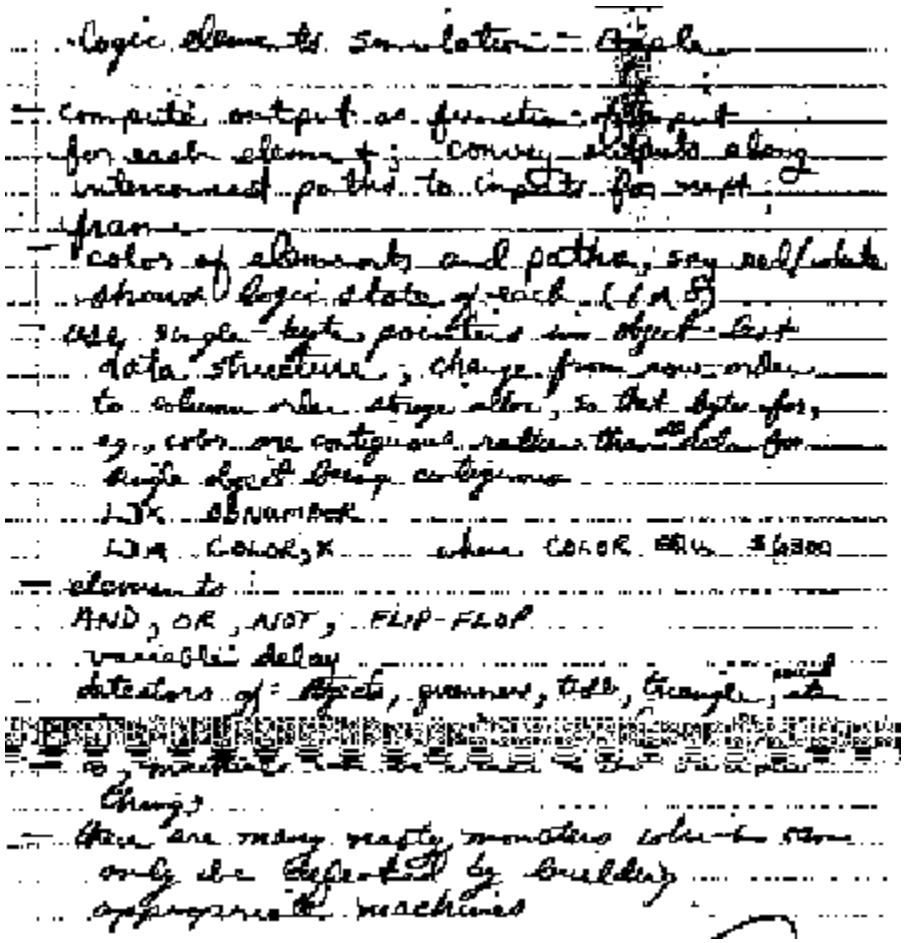

Figure 5-2

As an idea for a program is expanded into an implementation plan, the plan is, because of its brevity, vague or silent regarding many details. A programmer can easily get bogged down in these details (choosing variable names, addressing modes, etc.) if he begins writing the program too soon. A better method is to plan individual routines in the same way as the overall program was planned. The method I have evolved for doing this is to write down some of the major points to consider in short English phrases, and then outline the code to be written in a high-level English-like pseudo-programming language. For example, Figure 5-3 shows the details being worked out for the routine which writes the maze background on the screen, which was the first routine mentioned in the overall plan of Figure 5-l.

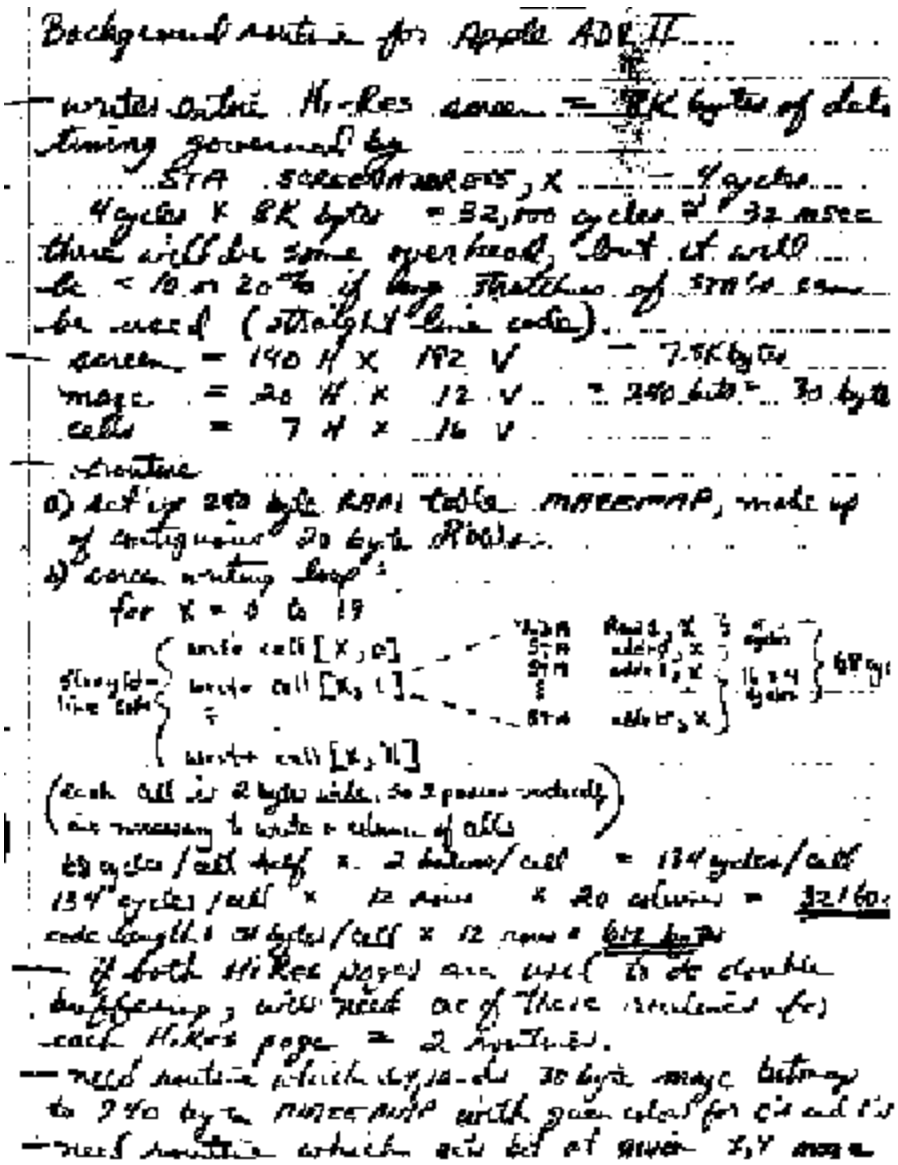

Figure 5-3

Once the general structure of the routine has been decided on, the programmer can concentrate on the many decisions to be made as he chooses the individual instructions that make up the routine. These low-level decisions include choosing which machine registers to use, choosing variable names, deciding when to add explanatory comments, choosing addressing modes, and choosing among various coding methods for implementing, for example, loops. Figure 5-4 shows the assembly language program created from the plan of Figure 5-3.

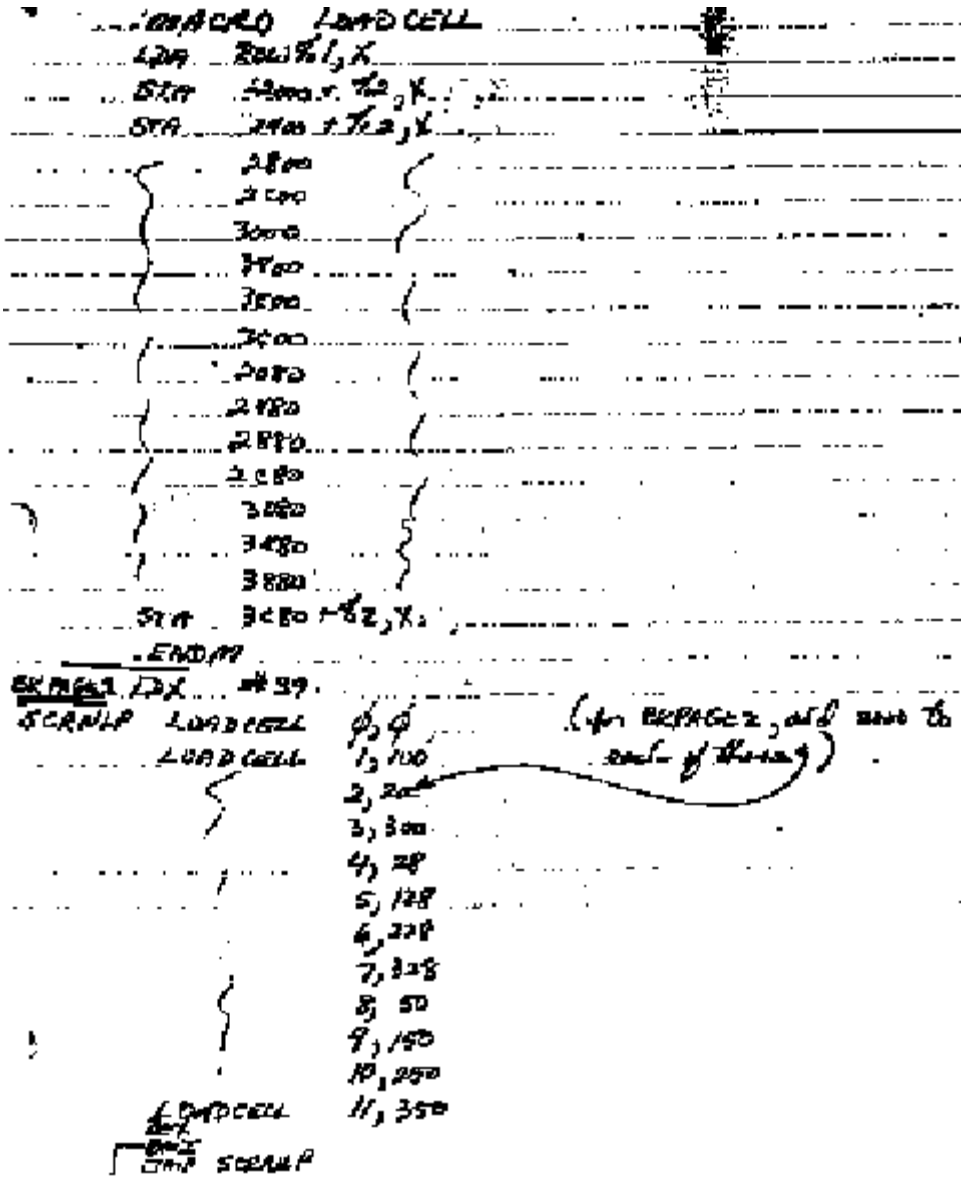

Figure 5-4

Advice to the Game Designer

As computer games rose spectacularly out of obscurity, there was little tradition to guide the designer of new games. Many people offered their opinions about what made a good game. Others moralized about what topics should or should not be addressed by computer games. It was difficult to make sense of all the uproar, and impossible to find any consensus. There were few, if any, agreed-upon principles of computer game design. Nevertheless, it is interesting to survey some of these often-conflicting bits of advice.

Some people offered the opinion that computer games were too violent. Usually in the next breath they added that computer games could more beneficially be used to motivate learning. Most of these people

were neither game players nor game designers. They seemed to regard computer games as a powerful, possibly dangerous force, which might, like nuclear energy, be beneficial if it was properly controlled.

When the critics said computer games were too violent, they meant that some computer games simulate violent, war-like acts such as dropping bombs and firing missiles. Disapproving of violence, they disapprove of its depiction, and fear that the young players of video games will fail to see the important distinction between a shoot-'em-up video game and pressing a button that destroys a real city. A revulsion for real violence -- for murder and war -- is sane. But simulated violence is not real violence. The kids in video game arcades know that their actions have no effect outside of those painted boxes.

Does participating in simulated violence give the participant a desire for or an insensitivity to real violence? A question like this is difficult to answer with certainty. All that could be charged is a subtle linkage of games to violence -- we know that kids do not walk, with glazed eyes, from video games to their fathers' gun racks and begin shooting. It might be useful to examine parallel cases. In the play, MacBeth murdered the kind of Scotland. Does Shakespeare's MacBeth breed assassins? A Monopoly player drives his opponents bankrupt. Does Monopoly educate tomorrow's heartless slumlords? The ethics and ideas and situations in fictitious worlds do affect those who enter them. But the effect might as likely be a realization of the awful consequences of an act as the imitation of its perpetrator. In any case, it would be absurd to ban MacBeth to prevent assassination, or to ban Monopoly to promote social justice. The link of video games to violence is equally remote.

Journalists photograph war. Their rationale is passing the truth on to others. Perhaps a complete bomber-game should convey not only the challenge of guiding a bomb to its target, but also the resulting horror of burning cities, vanished civilization, and cessation of life.

Sometimes a cautious publisher, to protect its reputation, can force a game to be modified to purge it of violence, or some other offensive quality. An early version of Rocky's Boots featured the kicking of ducks. Kicking ducks, cute furry little ducks, was considered violent, and violence was bad news for a publisher of educational software for young children. After protesting the absurdity of it, I changed the ducks into alligators, cruel scaly nasty alligators, thus lessening, or at least making more justifiable, the violence of kicking them.

A similar sanitization occurred during the development of Atari 2600 Adventure. The game was conceived as a quest, and so naturally the goal of the quest was to recover the Holy Grail. However, to avoid any possible offense of religious beliefs, the Holy Grail was given the safer name of the "Enchanted Chalice."

In another case, the game Magic Spells taught children spelling by presenting the letters of a word on the screen in scrambled order, and had a vocabulary of 500 spelling words. During a demonstration of educational software for some parents at a certain school, the spelling word "STRAIGHT" was randomly scrambled into "SHITTRAG." The parents were shocked and the teachers mortified. They felt that the program should have detected and censored profane letter-combinations.

Profanity filters have been tried in programs for children. A common practice in text-dialogue games has been to ask the player his name, and insert the name here and there to personalize the dialogue:

```
What is your name?
DANNY
  .
  .
  .
Good work, DANNY ! ! !
```

Mischievous 11-year-olds readily conceive the idea of giving a false name, and then giggle with delight when the program says:

```
Good work, Shithead ! ! !
```

A profanity filter routine can compare each input string typed in against a list of forbidden words, and refuse to accept profane inputs which it detects. This opened two lines of attack to the kids. The first was to discover the contents of the forbidden word list by trial and error, and the second was to circumvent the detection algorithm by such stratagems as "S-H-I-T-H-E-A-D."

At the same time that they were trying to remove the violence and profanity that might contaminate their children, many people felt that educational computer games were a good thing because the motivation of the video game could be allied with the wise choices of curriculum designers. Educational software could make learning fun! This missed the point. Learning was already fun, and interesting, and challenging. It was school that was boring, forcing children to go and sit in the same place day after day, with a limited variety of activities.

A real merit of using computers in schools, besides the possibilities of learning about computers themselves, was that a computer simulation could bring into the classroom a process that was too dangerous or too inaccessible to bring there otherwise. For example, a car-simulation which modeled carburetor, pistons, distributor, drive shaft, and so on could be used by children to understand how a car works. A real car is more interesting, but it is also greasy, bulky, expensive, and has dangerous electric shocks and spinning rotors. Or again, a real volcano is not suitable for classroom demonstrations.

A computer simulation can also expose the insides of a process, give invisible radiation visible markers, and convert the scale of a phenomenon in space and time to be suitable for human inspection and patience. Our perception of real phenomena can be extended, as if we possessed Superman's x-ray vision.

The designers and players of computer games have always been guided by their own perception of what is interesting and enjoyable, and haven't paid much attention to the ought-to's of the moralists. They prize a quality of a game called "playability" -- being fun to play. A playable game offers the player some interesting choices. The player should be able to improve his performance. A game should have some novelty or innovation to distinguish it from its predecessors. A game should be challenging. A playable game makes it hard for the player to accidentally get stuck in hopeless situations. There should be a means of offering the player variety from one game to the next.

There are several different ways to insure that successive games differ from one another. In action games, the player can't duplicate his actions precisely from one game to the next, so each game is different. A human opponent is unpredictable, and this gives a game variety. Another technique used to achieve variety in a game is to drive game events from a random number generator routine.

Random number generation is like rolling dice. Each roll, a number is produced, by chance, from a range of values -- the range 2 to l2 for a pair of six-sided dice. Simulating a dice roll on a computer is difficult because there are no chance phenomena inside a computer. The output of every program is determined by the input given to it. The best that can be done is to produce a series of numbers that jump around within the given range of numbers, like dice rolls do, and that have no discernible pattern. The output of a random number generator seems unpredictable. Random number algorithms are designed to produce each number in the output range equally often, and to make the values of successive numbers independent of one another. Random number generators produce a flat, patternless, featureless series of numbers.

The use of cards and dice show that the element of chance is important in (non-computer) games. Random number generation provides a similar mechanism in computer games -- an unpredictable automatic selection from among several alternatives. It is worth using sometimes. However, events driven by random numbers have a certain dullness to them. The player has no real hope of discerning any pattern in the events at all, because these algorithms are designed to be devoid of pattern. In the real world, by contrast, events are often unpredictable, but they are controlled by underlying mechanisms which, upon inspection, reveal patterns. (Normally friendly dog attacks -- ah ha! Rabies. Snowstorm in July -- ah ha! Eclipse.) Controlling game creatures with patternless random number routines deprives the player of the joy of discerning patterns and understanding something new.

As an alternative to randomness, the computer game designer has another method at his disposal. Simulation of a complex but consistent environment is a better technique than randomness for achieving variety in a game. For example, in Atari 2600 Adventure the bat moved in straight lines through the complicated network of rooms, occasionally encountering an object that made him change direction. This mechanism gave the bat a very complex behavior. The bat could fly past the player frequently for a while, and then disappear to another part of the game-world, and could even get stuck in a trajectory that repeated until the player interrupted it. This complicated and (somewhat) understandable behavior made the bat interesting. A bat which was randomly injected, once every minute or two, into the room with the player would have been much less interesting. Furthermore, since the player would have been unable to understand, predict, or control the bat's behavior, dealing with a random bat would have been much more frustrating.

It seems to me that simulation is a technique that offers material for designers of both educational games and games made purely for entertainment. Just about any phenomenon that can be named can be simulated. Fighting bull walruses. Snowflakes forming. A bolt of lightning. A moral dilemma. Given a phenomenon to simulate, the problem is to decide what are its parts, how these parts can be represented with numerical values, and what the relationships are that let these parts affect one another. Furthermore, since an interactive graphical simulation is an attractive use of the computer's resources, it is desirable to give the parts of the phenomenon a graphical representation, and to introduce a means for the player to interact with and participate in the phenomenon being simulated. When a graphical simulation faithfully mimics and exposes an inaccessible part of reality, it expands our perception.

Chapter 6

Spaces

A computer game consists of entities -- objects or creatures -- and the spaces they inhabit. The spaces are the medium upon which the action of the game occurs. A chessboard defines chess-space, the squarish path around the board defines Monopoly-space, and an ice-rink defines hockey-space. The spaces of computer games are portrayed upon the glass of computer display screens.

The action of the first video games was confined within a single display screen, but designers soon realized that the small screen could be a window into a much larger space. A shifting viewpoint allowed a user to inspect parts of a complex world in detail, and control of this shifting allowed him to move through world-space. The space viewed through the screen was sometimes two-dimensional, allowing a convenient mapping from the 2-D screen to a 2-D fragment of the space. With more difficulty, a three-dimensional space could be viewed through the screen, using the familiar convention of perspective that makes a flat landscape painting have "depth."

The manner in which the viewpoint moved through a space depended on the nature of the space. Scrolling was one of the first view-changes tried in video games, perhaps because scrolling text had long been a standard convention in editor programs. An alternative to scrolling was to connect screens edge-to-edge, as in Atari 2600 Adventure and Rocky's Boots, so that the view shifted from screen to screen in discrete hops. A third method of viewing a large 2-D space was to divide the screen into independent windows which showed different parts of it. For a perspective view into a 3-D space, the obvious view-transformations were moving the viewpoint through the space, and changing the direction and magnification of the view (panning and zooming).

Scrolling worked well for moving around in a plane, whereas room-to-room hops of viewpoint allowed wrap-around paths, one-way connections, and non-unique diagonal rooms. A network of edge-connected rooms could have a very strange interconnection pattern, as the mazes of VS Adventure illustrated.

Dividing the screen into independent, overlapping windows was an idea originally used in the user interface to the programming language Smalltalk. These windows were not really thought of as views into parts of a connected space. Rather, they were used as a means of simultaneously showing several related batches of information. The screen itself was the global space through which the user moved (with his cursor), with the corners of partially covered windows being the links to the contents of those windows. The metaphor was that the windows were sheets of paper on a desk, and a corner of a sheet could be grasped and the sheet pulled to the top of the pile. Windows were the sub-units of a space filling the screen, whereas in a network of rooms, the screen was a sub-unit of a larger, surrounding space.

Individual windows also had an internal spatial structure. Some windows scrolled through larger spaces of text. Windows full of graphics could shrink or grow. Parts of some windows could be activated, producing "pop-up menus," which might, in turn, contain their own subparts that would expand upon activation. A pop-up menu behaved like a Jack-in-the-Box -- a menu would appear, overlaying part of the window that spawned it, and then disappear after it had been used. Scrolling and pop-up menus made a window an entrypoint into a complicated information-space through which the user could move.

Zoom

The ability to zoom the viewpoint in to examine details in a small region of a space makes it practical for the space to contain details at different scales. Maps on paper do a good job of showing several levels of detail, considering the unzoomable medium upon which they are printed. A computer display offers the possibility of making interactive maps, which can pan and zoom. A zooming map could handle a much larger range of scales than a printed map. For example, a zooming map of the solar system could handle the million-fold scale difference between the orbit of Pluto and the orbit of Phobos, Mars's inner moon.

A dynamic map also offers the possibility of mapping things that move with respect to one another. Planets orbit the sun. Continents drift. Species spread and then die out. Birds migrate. Explorers wander. Embryos grow and develop. Proteins curl up as they are constructed. These processes can be mapped in time, as well as space. An interactive simulation of a process is probably preferable to a fixed-sequence "movie" of the process in action, but making a movie may be much simpler than making a simulation. A movie of the archetypal events in a process can be quite informative, particularly when the movie can be stopped, and run at different speeds forward and backward, and when the user can pan around to locate and zoom in on events of interest. Anyway, the sequence of events in some processes admits no variation. The script has already been written for continental drift on Earth.

For processes with events on widely varying time scales, time-zoom is necessary. This allows the process to be viewed at different rates. A movie of the first few minutes of the universe, starting with the Big Bang, needs time-zoom because events happen in trillionths of a second (and faster) at first, but rapidly slow down to a scale of seconds (and, later on, millennia).

A "conceptual zoom" is an idea proposed by artist Aaron Marcus. The name "zoom" was given to the magnification of real scenes with lenses. Its effect is the expansion of a region in an image, revealing details therein.
A conceptual zoom preserves the idea of expansion and emerging detail, but discards the requirement of magnifying real objects. A conceptual zoom can fade from one superimposed representation to another as scale changes. For example, a satellite photo of New York City could expand, giving way to a street map. The street map might, in turn, give way on expansion to architectural blueprints of individual buildings.

An enormous amount of data would be necessary to record the floor plans of all the buildings in New York City. A similar problem exists in every system that allows zooming and panning with computer-generated images. It is impractical to store complex images for every point that might be zoomed in on. (Although, if every architecture firm in the world stored its plans in data banks accessible through an Archi-Net, maybe . . .) One good solution is to generate images based on parameters stored for the area of the image being examined. Some phenomena like galaxies or atoms are sparse, with vast empty areas surrounding relatively few detailed objects. In other cases, many instances of a once-defined object can be placed, perhaps randomly, as details in a region of an image. For example, one grass-plant subroutine could provide all the detail for a vast prairie. The grass subroutine might have parameters allowing plants of different ages, so that rather than being an orchard of identical plants, the prairie could have both random and systematic variation in its vegetation. A specific grass-plant, in turn, might expand to reveal details dependent on the parameters from which that plant was generated.

A more abstract variety of conceptual zoom is the ability to "go inside" objects which appear on the screen. Going inside an object can be an abrupt change of view (to its interior) rather than a continuous magnification. Jumping inside an object is an abbreviated zoom, just as room-to-room motion is an abbreviated pan. Like panning or scrolling, continuous zooming requires a continuous space, like a plane. The view-jump of going inside allows movement through abstract, discontinuous spaces with many

levels of detail. For example, a computer is organized hierarchically -- logic gates and memory cells are combined into registers, adders, counters, memories, and processors. A user could move through this hierarchy, going inside the symbol for a complex assembly to examine its internal construction.

3-D

We live in a three-dimensional space. Our stereoscopic vision perceives objects as having three-dimensional positions. It is therefore a natural goal to simulate 3-D space through the window of the computer screen. However, adapting a 2-D screen to display a 3-D space offers some challenging problems. Furthermore, the commonly available pointing devices -- joystick, mouse, and tablet -- are two-dimensional, and so not really adequate for manipulating three-dimensional objects.

A perspective transformation can be used to project points in a 3-D space onto a 2-D one. This is like using a flashlight to make a flat shadow on the wall from a 3-D birdcage. The equations for implementing the perspective transformation are simple and well-understood. However, in general, current personal computers are not fast enough to generate complex 3-D objects for use in interactive programs. Several multiplication and division operations (which are slow on personal computers) are needed for every point in a 3-D object, whereas the display routines for 2-D bitmap objects can get by with the fast operations of addition and table lookup.

The surface of a 3-D object can be represented with many triangular facets, as on a jewel. The display routines for such an object must separate the visible facets from the hidden ones and calculate the color of each facet. In addition, it is desirable to give a texture to each facet, to simulate the highlights, reflections and shadows produced by light sources, and to smooth the object's faceted surface. All of these operations are very slow, taking hours on large computers to construct the images for a few seconds of animation.

The image on the screen of a computer game must be produced in a fraction of a second. Some games have achieved 3-D effects by restricting the position and rotation of objects in the game and of the viewpoint so that display routines can be simplified and thus be faster. For example, Zaxxon, an arcade video game, showed a climbing, banking, diving airplane. The point of view was behind, above, and to the side of the plane. The player could change the plane's altitude and lateral position as the terrain moved under it. On the screen, the terrain scrolled diagonally under the plane, and the plane moved up and down, or sideways at right angles to the scrolling terrain. Both the scrolling terrain and the airplane could be shown with (2-D) bitmap images, so that Zaxxon was able to give a very powerful 3-D effect using fast 2-D display techniques. It is important that the plane was confined to flying in a narrow corridor over the terrain -- as long as the plane stayed in this narrow range of positions relative to the viewpoint, the same image for the plane could be used at all of its positions. Without constraints on the plane's position, the player might fly up near the viewpoint, which would require a display algorithm for making airplane images of different sizes.

Flight simulation programs have to display the terrain under the aircraft from different heights and angles. Programs have appeared on personal computers which can handle this display task in real-time. Since, in a flight simulator, the point of view is out the window of the airplane, an image of the plane itself does not need to be displayed. However, if there are other objects in the simulation, like another plane to have a dogfight with, then it must be possible to view these objects from different angles and at different distances. It is fitting that flight simulation programs use joysticks for input since the computer joystick input device is descended from the joystick in early airplanes which was mechanically linked to the flaps of the plane.

A human has two eyes, and uses slight differences in the images falling upon his two retinas to judge the distance of objects in his field of view. Vision based on two side-by-side points of view is called stereoscopic. Using a single computer screen as a window into a three-dimensional world throws away the

benefits of stereoscopic vision, because both eyes see the same (flat) image on the screen. Systems which present separate images to each eye can give a very convincing impression of three-dimensional depth. Stereoscopy is used for viewing 3-D models of complex molecules, like proteins. A few arcade video games have tried it, too.

At the University of Utah in the early days of computer graphics, a wonderful idea incorporating stereoscopy was explored -- the head-mounted display. A lightweight stereoscopic display was built into a helmet for the user to wear, along with a device for sensing the orientation and position in the room of the helmet. Two images were generated, based on the positions and orientations of the two eyes, which were known from that of the helmet. This allowed an image of a three-dimensional object to hover in the middle of the room while the user moved his head (and the rest of himself) to examine the object from different angles. This 3-D object existed in the same space as the user. The head-mounted display offered interesting possibilities. An architect could look at a proposed building in its context of the city skyline, with him floating high above the city, and then he could descend and enlarge the building so that he could walk through it. With a 3-D pointing device, like the one that sensed the helmet's position, a user could grasp and deform the ephemeral objects that surrounded him. Thus, on the practical side, a user could sculpt models of parts to be manufactured.
On the artistic side, he could sculpt moving three-dimensional forms unfettered by gravity or structural materials. Two wearers of head-mounted displays could enter into the same universe of objects, working cooperatively on a project, or throwing fireballs at one another in a fantasy game. The promise of the head-mounted display has not been explored over the last fifteen years. Only a few have been built. Perhaps today's cheaper technology makes the head-mounted display a good solution in some area of industry or entertainment.

Adventure Game Real Estate

A computer network can allow several players to play in a common game even though they are physically separated from one another. The players communicate over the telephone lines through a central host computer. The game being played can be, among others, an adventure game full of rooms and objects. Several players can simultaneously inhabit the same adventure game world. This can work with any kind of adventure game, text or graphic, but the multi-player graphical adventure game is an interesting case to consider. A player of a graphical adventure game sees one room at a time on the screen. Two players in the same game world would see one another only if they were in the same room. Having two or more human-controlled actors in an adventure game allows for both cooperation and competition among the players. A band of cooperating players could explore an unknown labyrinth, in the style of Dungeons and Dragons. Two players (or two tribes of players) could fight, stealing objects from one another, having tugs-of-war over important weapons or treasure, and throwing things at each other. Algorithmically-controlled creatures could be indistinguishable from human-controlled actors. Since all the players and creatures in the game would be represented on the screen by arbitrary small colored shapes, the game would be like a masquerade ball, with everyone anonymous.

A computer network can have dozens, or hundreds, or thousands of users connected to it simultaneously. Several of these people could agree to play together for a specified time, after which the game would be over. However, another possibility is to have an on-going adventure game, an evolving scenario into which people could enter when they logged on to the network. In an on-going game, events would be occurring continuously as different players entered and left the game, and as the creatures in the game responded to the situations they encountered. It would seem that after a player had gotten a valuable treasure, he would need to stay connected to the network 24 hours a day in order to protect his new possession from the predatory hordes of players and creatures in the game. Clearly, a habitual player of an on-going adventure game would need a secure place to store his gear -- his weapons, armor, treasure, maps, tools -- while he was gone from the game-world.

Imagine that the adventure game world was a giant checkerboard of rooms -- a thousand screens by a thousand screens. Each player could have a particular room as a stronghold in which to store his stuff and know that it would be safe while he was logged off the network. The player would, in effect, own that room. This room would be the player's bank vault, his castle, his inner sanctum. Ownership of such a vault-room would be important to a serious player in this game world. In addition to the usual network charges, it is conceivable that a player might pay real money, say 50 cents, for ownership of a vault-room. For the entrepreneurially inclined, this presents an interesting opportunity: create an imaginary world and then sell real estate in it.

An adventure world full of empty rooms might not be worth much, but as players began to build their castles here and there, to connect secret magical passages between distant castles, to build pretty-looking facades at castle entrances, the real estate might begin to appreciate. Certain areas might develop a concentration of dwellings -- that is, become a town. The few remaining empty screens within the area of a town might acquire some value, based on their location. Owning some space in an adventure world might be worthwhile because of the things that had come to go on there, the players or creatures that hung out there, or the traffic that passed through.

Real estate derives its value not so much from its intrinsic ability to support a building as from its context of surrounding property. A lot's nearness to desirable places like parks, offices and schools strongly affects its value. The marshy land along Buffalo Bayou is valuable not because of its soil or vegetation, but because it is surrounded by the rest of Houston. An imaginary world could acquire some value from the structures built within it, and the activities and commerce going on there. In fact, since more independent worlds could be created fairly easily, the inhabitants of a world and the things they had built would be the entire wealth of such a world.

Player as Perceiver

Philosophers ask the question "Is the physical world to any degree dependent on a perceiver for its existence?" Or, to cast the problem in a specific situation: "Would a tree falling in the forest make a sound if there was no one there to hear it?" These questions have always seemed rather ridiculous to me. However, in a simulated reality in which every detail seen by the viewer is explicitly generated, the question's significance is easier to understand. Can events occur, unseen, in parts of a simulated world beyond the viewer's direct observation? This really is a question of consistency. That an unobserved event occurred must be inferred from its observable consequences. The bat in Atari 2600 Adventure moved objects around regardless of whether the player was there to observe it. Evidence of the bat's actions could be seen later in the rooms where these actions occurred.

In a sense, a computer game for one player is a universe created for one perceiver. Without the player-perceiver, the game-universe would not exist. In a galaxy simulation program, stars could be randomly generated to surround the player as he wandered through the galaxy. Consistency would demand that on returning to the same part of the galaxy, the same configuration of stars should be there. To model a galaxy like our Milky Way, which contains l00 billion stars, it is clearly impractical to record data on position and brightness for each individual star. Stars must be somehow described in the aggregate. A scheme of random star-generation based on densities in various regions of space could repeatably generate the same stars for a given region of the galaxy, thus solving the return-visit-consistency problem. All l00 billion of these stars would exist, implicitly, in the star-generation algorithm, but only a small fraction of those could ever be visited, explicitly, by a player of the game.

Most computer games, up to the present time, have populated their spaces with a limited number of objects. Each object required a few bytes of memory to describe it, so the number of objects was limited by the amount of memory available. Adventure games, for example, used tables to describe the position and properties of the various treasures, weapons, and tools in the game. An adventure game could contain

hundreds of table-defined objects, but not millions. In contrast, an algorithm which used a position in space to generate the characteristics of the objects in that region could define millions or billions of objects.

In the case of both table-defined and algorithm-defined objects, an input number -- an object number or a position -- was accepted and some output data was returned describing the properties of that object or of the object at that position. For example, using the number "5" to access the object tables of an adventure game produced the properties of object number 5. In the galaxy simulation, the coordinates of a sector of space could be used to calculate the characteristics of the stars in that sector. The difference between the two methods was that the table required some memory for each possible input value, whereas the algorithm didn't. Thus, the table method was restricted to a limited range of input values -- a few hundred or a few thousand -- but the algorithm method wasn't.

Coordinates can be thought of as a way of numbering all the sectors in a galaxy. The star-algorithm takes a sector number and produces a description of the stars in that sector. A sector can be treated as an object, and the stars within it as characteristics of that sector. Thus, this descriptive method assumes that each possible object has a number, and that the object's characteristics can be described as a function of that number.

This idea of connecting the coordinates in space with an enumeration of objects allows a space to be populated with an immense number of objects. An enumeration of the l00 billion stars of the Milky Way would require about 27 bits, which is slightly less than 4 bytes of memory. An enumeration of all the cubic inches in a cube l00,000 light-years across (just the right size to enclose our galaxy) would require l8 bytes. Neither 4 nor l8 bytes is too large to serve as input to a routine which generates characteristics for the objects at each position.

An algorithm-populated space seems to solve the problem of providing detail for whatever region of a space a viewer chose to zoom in on. However, some problems present themselves. Zooming demands consistency between close-up and distant views of the same region. Considering two views of the same region which differ a millionfold in scale, it is impractical to search through all the details of the small view and its neighbors in order to construct the large view. There must be a way to extract the prominent details of a large-scale view without scanning all the details of its sub-views.

Although there are problems to be solved, it seems that it may be possible to construct a simulation of a galaxy in which the viewer could zoom in on any part of it, zooming down from the spiral-shaped entirety, through clouds and clusters of stars, to finally center upon one growing star that fills the screen. Such a galaxy model might have many identical or nearly identical stars scattered through it, concentrating resources upon the description of the spatial distribution of stars. Or, alternatively, it might generate many different types of stars, with various sizes, temperatures, ages, and compositions. Having zoomed in on one particular star, it would be possible to use the characteristics generated for that star to drive a planetary system model, which assigned likely distances, masses, and compositions to planets based on the properties of the parent star. As with the galaxy model, it should be possible to zoom down from the entire planetary system to view any individual planet. The star-system model would have assigned certain characteristics to the planet. In turn, these characteristics could drive a planet model, which generated surface features like mountains, craters, atmospheric flow patterns, and storms.

Using details generated in one model to drive a second model would be called cascading the models. This cascading can run through several levels. There is no reason that the galaxy model could not be driven by a simulation of the universe, populated by clusters and clouds of galaxies. Thus, by cascading models of the universe, galaxies, stars, and planets, a simulated universe could be constructed. This one little program would contain too many planets to ever be fully explored in all the future history of humanity.

Furthermore, fanciful universes could be constructed that contained more details than there are particles in the real universe. (The universe contains l088 electrons, which can be enumerated with a mere 37 bytes.) However, there is a difference between detail and complexity. A space can be designed with an infinite number of details -- for example, a fractal into which one could zoom perpetually. But in the fractal new details repeat the old. Complexity means diversity, variety, new stuff. The complexity of a

simulated universe is limited by the number of clauses and provisions in the simulation program. A simulated universe could never approach the complexity and diversity of the real universe, which has many surprises waiting for us. However, the universe designer should not be too disheartened to learn that he can create universes that, although being more repetitive than the real universe, can have more details in them.

Chapter 7

Creatures

Computer games often contain creatures, active entities that move around and do things. The behavior of creatures is modeled on that of animals, which prowl around, choosing to eat, mate, or fight based on their perceptions of their environment. A computer simulation of a creature must explicitly model the creature's sense and actions, and the rules which connect the two. (This three-part structure is similar to the input-simulate-display main loop of a game program.) A creature's senses detect conditions in its world, and thus a sensing routine must test the data describing the states of objects in the game world. For example, in Atari 2600 Adventure a dragon (or bat) sensed which other objects were in the same room with it, and it also sensed the type of each of these objects. An action performed by a creature changes its own state or the state of some other object, and thus changes the underlying data which defines these states. When a creature moves itself, for example, it changes the coordinates that describe its own position. The rules for triggering actions by senses define the behavior of a creature. The dragon in Adventure is a good case study in rule-governed behavior.

Chasing and Fleeing

The dragon chases some objects (like the player's cursor) and flees from others (like the sword which kills dragons). The essence of the chase algorithm is that the coordinates of the pursuer are compared with the coordinates of the prey, and the results -- greater than, equal, or less than -- tells whether each component of the pursuer's velocity should be positive, zero, or negative. If the roles of pursuer and prey are reversed, then the algorithm calculates a path for the creature away from the other object, not towards it -- a flee algorithm. Figure 7-l compares chasing and fleeing. If the prey moves while being chased, then the path followed by the pursuer can be quite complicated, even though the algorithm is simple.

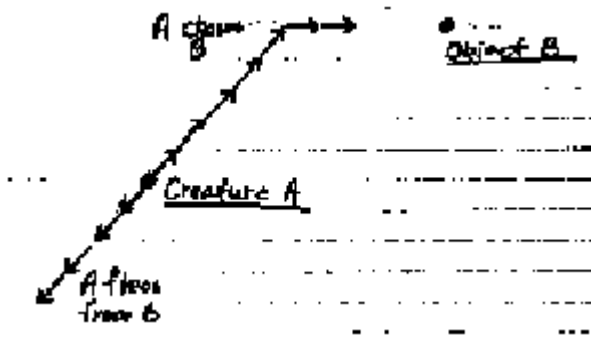

Figure 7-1

Chasing and Fleeing

A creature can chase after or flee from any object. The dragon might, for example, flee from the sword at one time, and pursue the chalice at another time. But what happens when both the sword and chalice are in the same room with the dragon? The dragon must choose among (or perhaps make a synthesis of) its various inclinations. The solution used in Atari 2600 Adventure was, for each creature, to have a prioritized list of objects to which the creature responded. The creature's behavior is governed by the highest object on the list which is in the same room with the creature. Each creature has its own chase-flee list. The priorities of the three dragons (and hence their behaviors) are different. The bat of Adventure has a similar priority list. (See Figure 7-2.) The pursuit of an unmoving object quickly results in the pursuer sitting directly on top of the object, waiting there until an object of higher priority (like the player's cursor) enters the room to lure the guardian creature away.

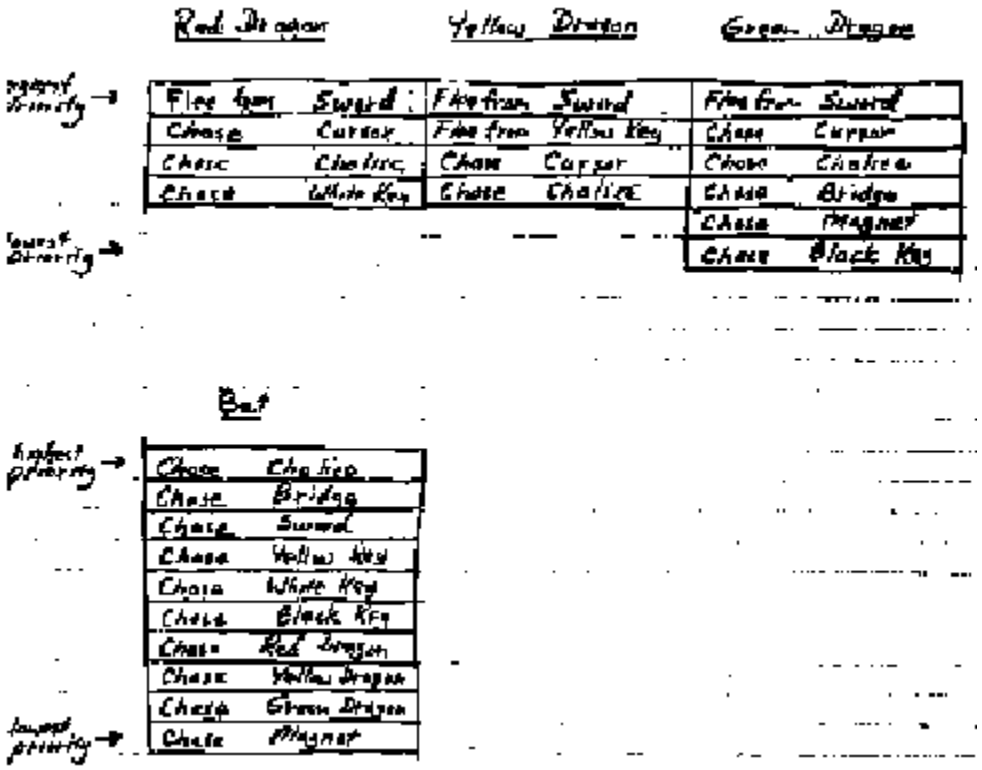

Figure 7-2
Chase-flee lists for the creatures
in Atari 2600 Adventure

This scheme for controlling the actions of a simulated creature is reminiscent of a school of psychological thought called Behaviorism. According to this theory, living creatures respond to certain stimuli, so that the entire behavior of a creature can be specified by a set of stimuli, and the corresponding responses:

Stimulus 1 → Response 1
Stimulus 2 → Response 2
. . . . . .

Stimulus N → Response N

The algorithm using the chase-flee priority list meshes pretty well with the Behaviorist model. The presence of various objects in the same room with the creature is the stimulus; pursuit and flight are the responses. That the dragon is credible as a creature in the simple world that it inhabits suggests that its behavioristic instincts are a model worth considering for simple creatures. That an animated simulation of a creature in an interactive game provides a forum for testing the ideas of behaviorism suggests that video games might be a good foil for testing other psychological theories, and for explorations in artificial intelligence. Creatures could have goals, formulate plans, and try to execute them; or they could be moved from one emotional state to another; or they could contract friendships and hold grudges. Creatures like these could not only provide interesting characters for animated adventure games, but they could also exercise and test the models being simulated of planning, emotion, learning and knowledge.

Simulated Behavior

Plans, emotions and bonds of friendship are internal mental states. A program which models these things simulates a creature's brain. The structure of a brain, human or otherwise, is a fascinating subject. The field of artificial intelligence attacks this topic, but has fragmented itself, seeking to closely emulate various human abilities such as vision, use of language and problem-solving. Researchers are still immersed, after 30 years of effort, in discovering the mechanisms by which vision, language, and choice work. A machine simulation of a human-like intelligence, integrating these various abilities, is decades away. However, not all brains are so complex as a man's. Earthworms have brains, and even amoeba respond to stimuli with actions. It might be interesting and informative to start with simple models of creatures acting in simple universes (like the dragons in Adventure) and build up creatures with more complex behaviors from that foundation. Certainly, simple models of plans, emotional states, and knowledge could be defined. A creature's plans, emotions, or knowledge would affect its behavior. A plan is a sequence of actions to be carried out, with the intent of achieving some goal. An emotion is a state which affects what goals are chosen, with, for example, fear eliciting flight, and desire eliciting pursuit. Knowledge is a list of objects external to the creature, and their attributes and relations. Perhaps it is too ambitious to have tried to simulate the behavior of a man, a creature of 50 trillion cells, without first simulating the behavior of a one-celled amoeba.

Research into the behavior of E. Coli bacteria has turned up some interesting parallels with the dragons of Adventure. These bacteria have spots on their cell membranes which can respond to different concentrations of nutrients or toxins in their immediate surroundings. They also have flagella, whip-like appendages which can spin around to push them through their fluid medium. The behavior of a E. Coli seems to be that when conditions are improving, that is, when the concentration of nutrients is increasing or toxins decreasing, then the bacterium will motor straight ahead for long periods. But when its sensors show that it is moving into a less friendly environment, the flagella reverse their direction of spin, throwing the bacterium into a tumble which leaves it pointed in a new direction. Thus, the E. Coli travels randomly in various directions, but travels further in directions in which its lot seems to be improving. This behavior is effective in moving the bacterium out of regions with high toxin concentration, and into regions full of nutrients.

The behavior of the E. Coli is even less sophisticated than that of the simulated dragons. Rather than choosing in which direction to move, the bacterium is given a direction, and must decide if it is a good one or a bad one. In both cases, however, approach and avoidance are the responses to a set of stimuli from the surrounding environment. Also, both the dragons and bacteria require a mechanism -- judgment -- for deciding what to do in the face of conflicting stimuli. For example, what should the bacterium do when poisons and food are both becoming more concentrated? What should the dragon do in the presence of both a desired and a feared object? The mechanism of these decisions in the E. Coli is not yet known. The

dragon of Adventure ignores all but its highest priority stimulus. More complex behaviors are possible, for example, by synthesizing an action from the responses to two normally separate stimuli. Advanced creatures sometimes "kill two birds with one stone."

Creatures, both simple and advanced, are often confronted with forces in their environments which they cannot control. Creatures can at least have knowledge of these phenomena, or instinct, which is a kind of hereditary knowledge. Knowledge or instinct guides a creature to safe behaviors in hostile situations. In complex creatures, knowledge and understanding can render menaces harmless. Birds have little to fear from bears. Man has conquered fire and controlled disease. However, not all phenomena are accessible to a creature's understanding. An E. Coli bacterium is not likely to comprehend the habits of the scientist who dabs nutrients and poisons into the dish it inhabits. Ants are not likely to understand the bulldozer that demolishes their nest. Similarly, certain mysteries confront human creatures. Why does the universe obey simple laws? Where can happiness be found? Why does death exist? Does time go on forever? Mystery beyond comprehension is more the province of religion than science. The unknowable is God.

Simulated creatures in computer games can interact with player-controlled entities, like the cursor in Atari 2600 Adventure. The player may have about the same capabilities and perceptions as the creatures, or alternatively, the player can be omniscient and omnipotent in the context of the game. He can play God. Depending on his personality, he can cultivate a universe of cringing and servile subjects, or a universe of jolly frolicsome creatures, coddled and protected by their creator. In either case, the absolute power of the human playing the game is unquestioned. Computer games like this can be created, provided that convincing simulations can be constructed of creatures with personalities, knowledge, and emotions. The creator of an imaginary world can be the God of that world. But lest we feel too powerful and mighty, it should be observed that we have no way of knowing that we are not ourselves being manipulated, casually and capriciously, in some greater being's imaginary world.

Appendix A: Program Structure of *Adventure* and *Rocky's Boots*

Getting a good idea for a computer game is only half the battle -- the idea must then be implemented as a working program on some computer. Thousands of decisions, major and minor, face the programmer as he constructs his game. Often, to make the game work at all, the programmer must arrive at certain insights as to how to structure his program. Thus, the design of the game program itself is a crucial part of computer game design. It is instructive to examine the structure of real game programs, to see how they have been constructed. As examples, the program for Atari 2600 Adventure and Rocky's Boots are worth looking at.

First, however, a few general observations can be made about computer game programs. An animated computer game lets the player manipulate an input device such as a joystick, and thereby affect the action on the computer's display screen. The action on the screen usually involves discrete entities (such as balls, spaceships, or monsters) that interact with each other. Thus the game program must get input data from the player, use that data and the current state of the game to simulate the behavior of all the entities in the game, and then show the player what is happening with an image on the display. This input-simulate-display loop is repeated many times a second. Also, to make the program into a game, there must be a test to see if some criterion for winning has been met. Figure A-l shows the general flowchart for an animated computer game.

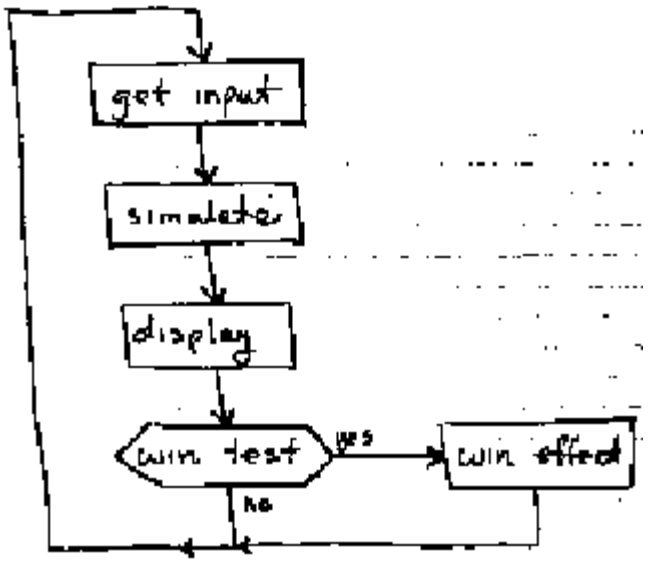

Figure A-1
Flowchart for a computer game program

There is a natural order in which to write an interactive graphics program. My habit is to write the display routines first, since their behavior can be tested by watching the screen. Second, I write input routines and use input from the joystick or computer keyboard to drive the display routines. From this early stage onward, the programmer can test much of his new code in the role of a player of the game, manipulating the joystick and checking that the response on the display is as expected. This is much faster and more satisfying than traditional debugging methods which involve peering at columns of numbers. Computer game programs have a nice property: all bugs are visible. If you can't see, it's not important.

A very interesting and important part of the program for an adventure game is the data structure which is used to represent the rooms and objects in the game. The rooms are interconnected into a network, and each object has a position in a room. The rooms and objects possess certain attributes, like color and shape. The data structure (see Figure A-2) numerically represents the properties and positions of the various rooms and objects. The room-list and object-list contain blocks of data that define the position, shape, and properties of each room and each object. The straightforward data structure of Figure A-2 is an idealized one -- in practice, various implementation problems and machine quirks complicate things.

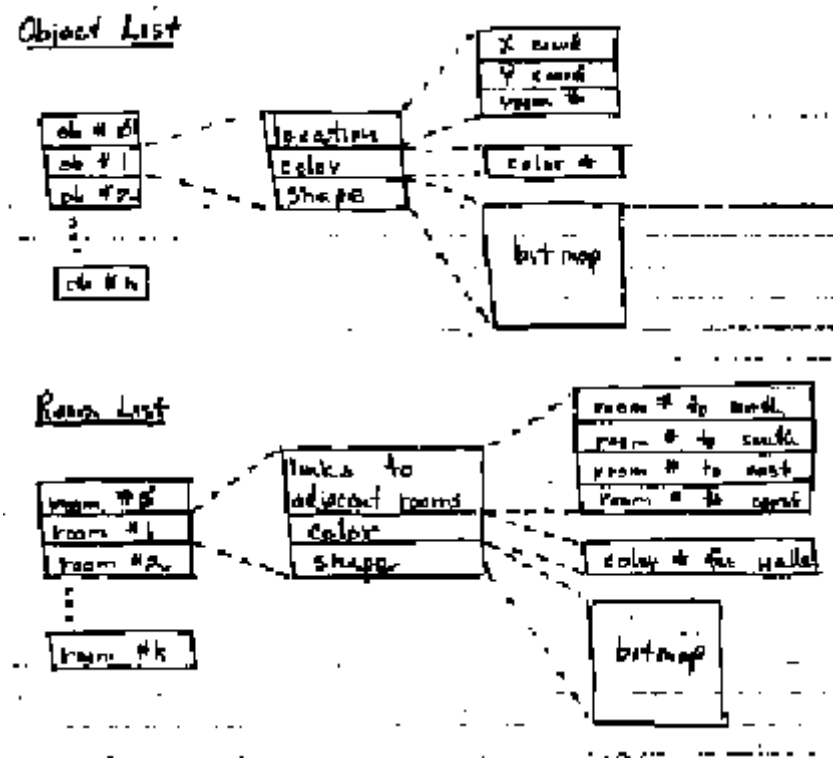

Figure A-2
Adventure game data structure

Program Structure of Atari 2600 Adventure

The data structure for Atari 2600 Adventure is shown in Figure A-3. The data for a given object was not all grouped into one block of memory. Rather, pointers were used to link several memory areas together. There were two reasons for this: first, the part of the data that could change as the game progressed (for example, object position) had to be located in RAM memory, whereas most of the data was in a different region of memory, in ROM. Second, some of the data was of variable length (object shape bitmaps), and therefore could not fit into the regular, fixed-size list structure without wasting space. Another complicating factor was that the data structure was set up to allow the shape of objects to be easily changed. (This allowed the bat and the dragons on the game to be animated.) Also, each object had an additional attribute (takability) specifying whether it could be picked up by the cursor or not. The bitmaps which defined the shapes of rooms and objects had some quirks. The object bitmap had to end with a zero byte. The room bitmap had two special bits which turned on or off the thin walls on either side of the screen. Most of the peculiarities of this data structure resulted from attempts to conserve memory, which was very limited on the Atari 2600 machine -- it had l28 bytes of RAM memory and about 4000 bytes of ROM.

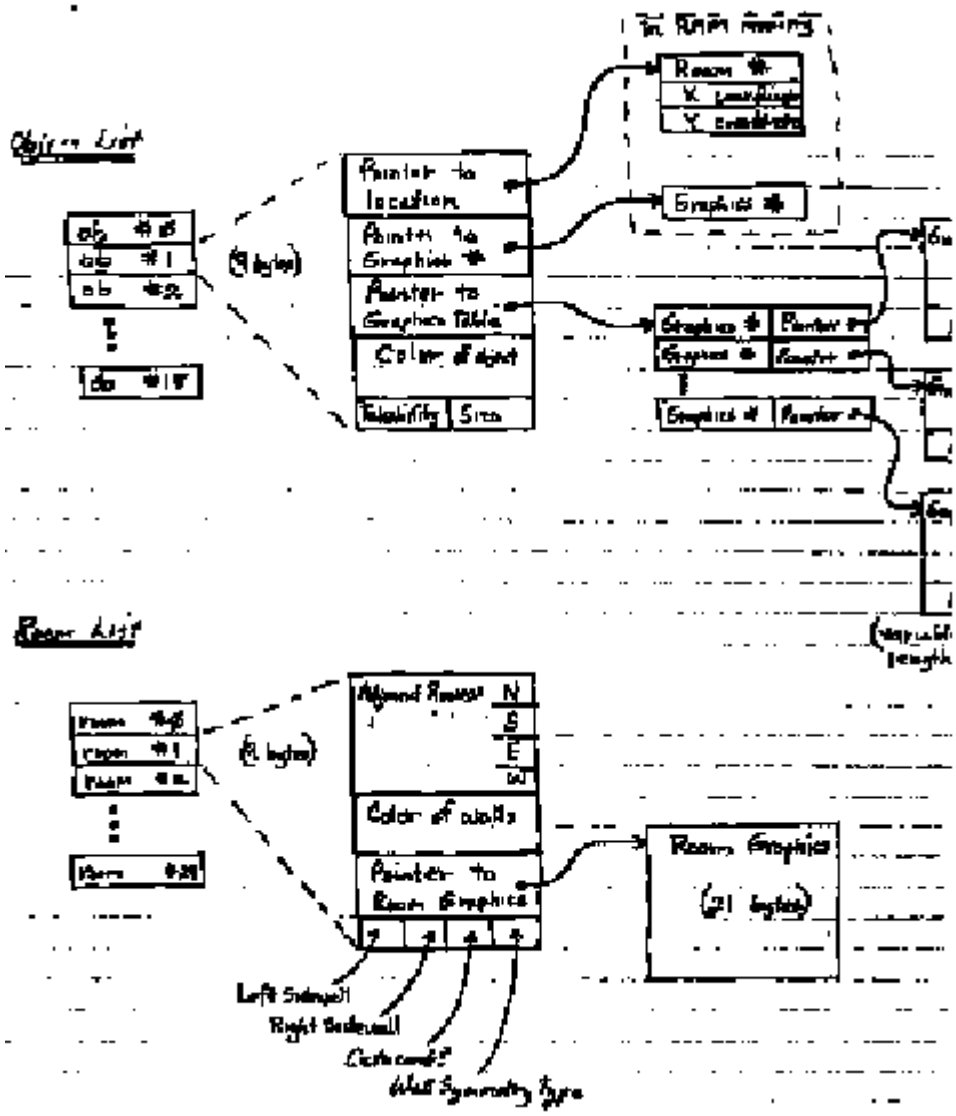

Figure A-3
Data structure for Atari 2600 Adventure

The program for Atari 2600 Adventure operated upon the room and object data. (See Figure A-4.) Input from the joystick controlled the movement of the player's cursor, within a room or from room to room. The cursor was treated as an object. Joystick input also controlled the picking up and dropping of objects. When the cursor dragged another object around, the data describing that object's position was changed.

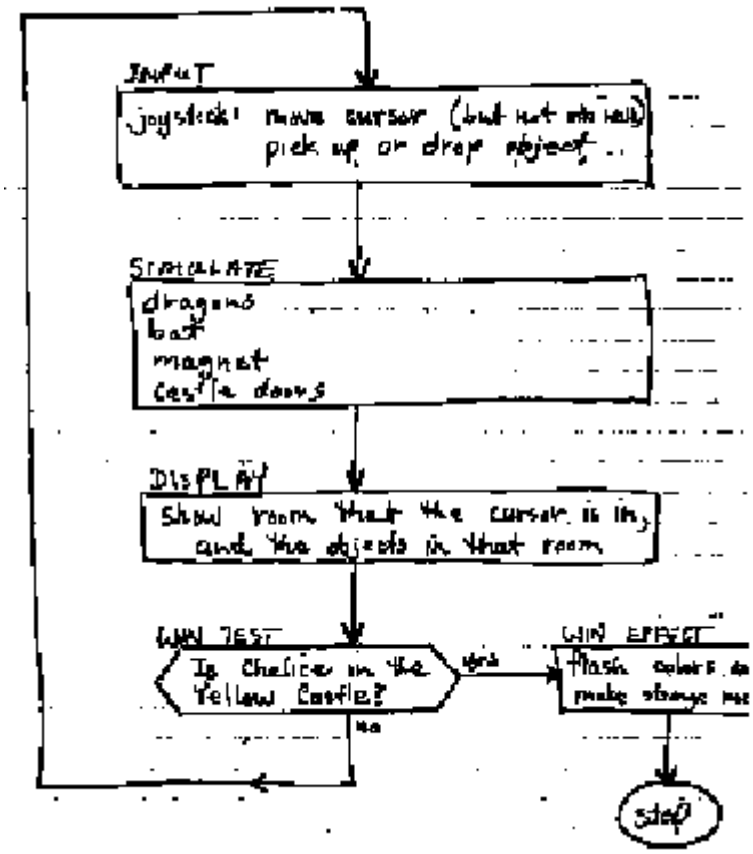

Figure A-4
Flowchart for Atari 2600 Adventure

The simulation part of the program defined the behavior of several types of objects: the dragons, the bat, the magnet, and the castle doors. During one iteration of the main loop of the program, corresponding to one animation frame, a subroutine was invoked to control the behavior of each of these active objects. One routine was needed for each type of object. There were three different dragons in the game. Each one was controlled by the same dragon subroutine, although different parameters were supplied to the subroutine (dragon speed and which object to use as the dragon) so that the different dragons could have different characteristics. The dragons and the bat were implemented as objects controlled by subroutines, but since they moved about on their own in the game, and initiated actions, they could be considered to be simulated creatures.

The display part of Adventure showed on the TV screen the room currently occupied by the cursor, plus any objects that happened to be in that room. The data in the object- and room-lists describing all the other rooms and the objects in those rooms was not used by the display routine. Nevertheless, the simulation part of the program diligently updated the positions and other attributes of those invisible objects. Thus, events could occur unseen, in other parts of the game world.

Program Structure of Rocky's Boots

The idea of rooms and objects and the data structure which represented them was the foundation upon which Rocky's Boots was built. The data structure for Rocky's Boots used the same fundamental organization as Atari 2600 Adventure -- a list of data blocks for rooms and objects -- but some improvements and additions were made to the new version of the data structure. (See Figure A-5.) Provision was made to connect different objects together, so that they moved around as a unit. Text strings were added as a new entity to the data structure, on an equal footing with rooms and objects. Two changes were made for efficiency in accessing the data. First, a linked list was maintained of all the objects in each room. Second, the order of the data in memory was reorganized, grouping the data for specific properties (like color) into memory blocks, rather than, as before, grouping all the data for a given object in one block of memory.

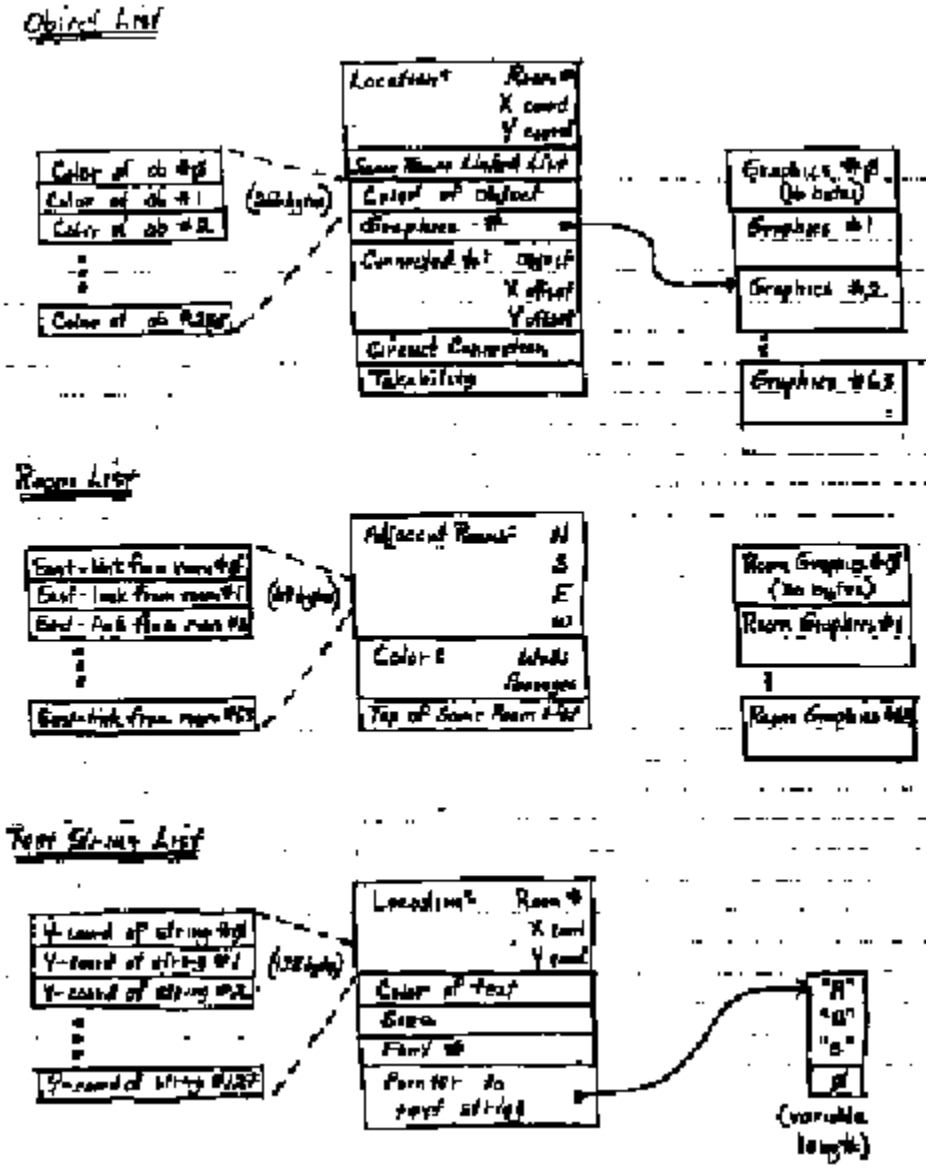

Figure A-5
Data structure of Rocky's Boots

One reason that greater efficiency was needed in the new version of the data structure was that the new host computer, the Apple II, had twelve times as much memory (48K versus 4K) as the Atari 2600, and thus could represent many more objects and rooms. However, this larger quantity of objects needed to be processed in roughly the same amount of time as before, namely, in one animation frame. Therefore each individual object needed to be processed faster. Atari 2600 Adventure had 30 rooms and l5 objects, whereas the data structure on the Apple allowed 64 rooms, 255 objects, and l28 text strings.

The ability to connect several objects together into a rigid body had some advantages. The bitmap used to represent the shape of a single object confined that shape to a limited size (7 by l6 pixels on the Apple), and "gluing" several objects together could overcome this size limitation. In addition, each object was defined to have a single uniform color, so a multi-colored object could be constructed by gluing together several ordinary objects. The logic gates in Rocky's Boots were each made up of several glued-together objects. Each input or output of a gate was a single object, and thus it could change color independently of the other inputs and outputs, and of the other objects making up the body of the gate. This was convenient for showing the state (0 or l) of signals feeding into the gate.

This interconnection of objects was implemented by slaving the position of one object to another (its master) with a fixed X, Y offset. When the master object moved, the position of the slave object was calculated as the sum of the offset and master's position. The position of all the slave objects was updated each animation frame, so that the slaves maintained a constant relative position to their masters. This property of connectivity was defined in the data structure by adding three new entries for each object in the object list: a pointer to a master object (possibly null), an X offset, and a Y offset. A null pointer meant that the object had no master, and could be positioned independently.

In Rocky's Boots , the master-slave scheme was used to interconnect the objects making up an individual logic gate, but a different method was used to hook up gates into circuits. When an output object was plugged in to an input object, this connection was recorded in the data structure as two pointers linking the objects to one another. These connection pointers served two functions. They were used for conveying signals from gate to gate as a signal propagated through the circuit. When the circuit was moved, they were used to position the gates in the circuit relative to one another.

The program for Rocky's Boots, like Atari 2600 Adventure, had a main loop composed of input, simulate, and display routines, plus a test to see if the player had won the game. (See Figure A-6.) The input and display parts of the program are similar to the corresponding routines in Adventure, although "plugging together" has been introduced as an input-controlled action, and text strings have been introduced into the display. The simulation part of the program is entirely different from that of Adventure, simulating signals moving through a circuit instead of monsters that chase the player around. (Well, there is one nasty little alligator in Rocky's Boots.) The criterion for winning the game is different, too.

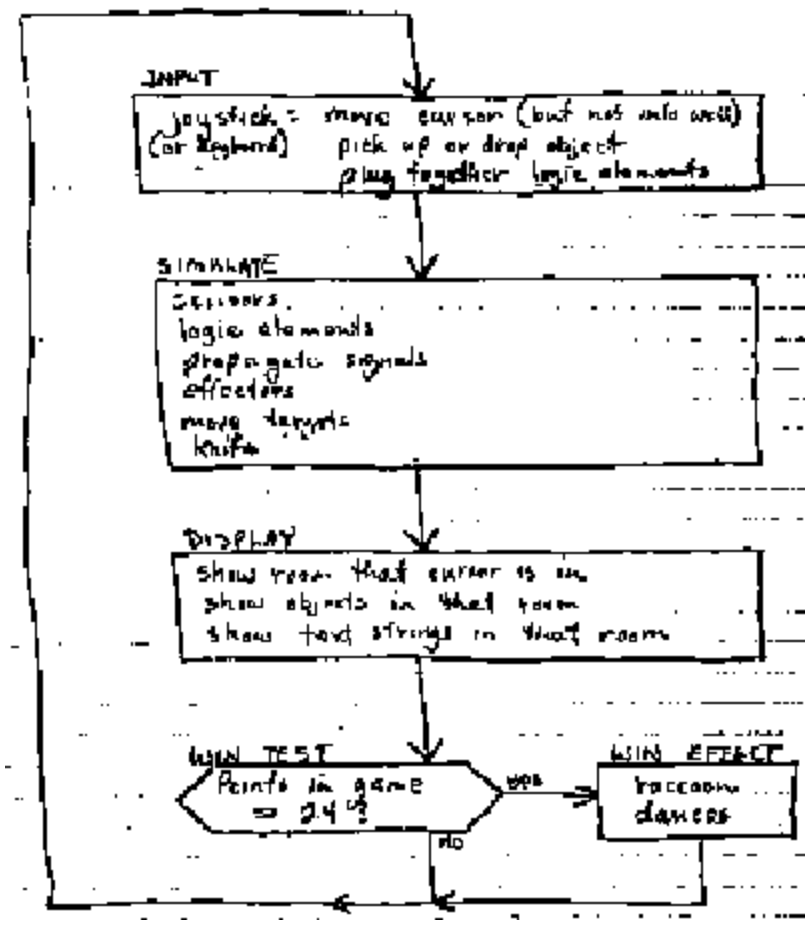

Figure A-6
Flowchart for Rocky's Boots

Most of the simulation part of the code in Rocky's Boots is concerned with simulating the various parts of the logic circuits in the game: the sensors, logic elements, and effectors.

The sensors are circuit components which, like human senses, produce signals in response to certain conditions. A green-sensor, for example, tests for the presence of a green object in a region near the sensor. The logic elements come in several types: AND gages, OR gates, NOT gates, flipflops, delays, and wires. Each of these logic elements calculates the state of its outputs based on the signals that have come into its inputs. The colors orange and white are used to how the state of each circuit component on the screen, and, in fact, the table in memory recording the color of each object is also used to signify the state (l or o) of each input and output. The effectors, like the boot and clacker, are triggered by their inputs into causing actions within the game world. The behavior of each type of component was controlled by a particular subroutine.

The logic circuit simulation works in a two-step cycle: simulate logic and propagate signals. First, the outputs of all logic elements are calculated from their current inputs, the outputs of the sensors are determined by testing the situation in the game world near the sensor, and the effectors are activated based on their current inputs. In this step of the cycle, all the circuit components operate independently of one another. Then, second, all output signals are propagated across the circuit connection to drive the inputs of other circuit components. These two steps happen once each iteration of the main loop, followed by the

display routines, which show the state of each input or output on the screen. Thus, a signal takes one animation frame to proceed from the input of a component to its output. Another way of saying this is that circuit components in Rocky's Boots have a one frame propagation delay.

The data and the program for both Atari 2600 Adventure and Rocky's Boots defined the entities being simulated in the games. Each room and object entity had its own private data area in memory, describing its state, position and properties. Each active entity in the game was controlled by a subroutine which regulated the entity's behavior, and detected and supervised interactions with other entities. The input routines allowed the player to control an entity (the cursor), and thus make things happen in the game. Both programs defined imaginary worlds, whose inhabitants and laws were completely defined in a few little pieces of silicon.

http://www.warrenrobinett.com/inventing_adventure/
